DINEI ANDRÉ ROCKENBACH

NADINE ANDERLE

ANÁLISE E AVALIAÇÃO COMPARATIVA DO DESEMPENHO DE BANCOS DE DADOS NOSQL

Três de Maio

2017

DINEI ANDRÉ ROCKENBACH

NADINE ANDERLE

ANÁLISE E AVALIAÇÃO COMPARATIVA DO DESEMPENHO DE BANCOS DE DADOS NOSQL

Trabalho de Conclusão de Curso apresentado à Faculdade Três de Maio para obtenção do grau de Bacharel em Sistemas de Informação da SETREM

Orientadores:
Dr. Dalvan Jair Griebler
M.Sc. Samuel Camargo de Souza

Três de Maio

2017

TERMO DE APROVAÇÃO

DINEI ANDRÉ ROCKENBACH
NADINE ANDERLE

ANÁLISE E AVALIAÇÃO COMPARATIVA DO DESEMPENHO DE BANCOS DE DADOS NOSQL

Relatório aprovado como requisito parcial para obtenção do título de **Bacharel em Sistemas de Informação** concedido pela Faculdade de Sistemas de Informação da Sociedade Educacional Três de Maio, pela seguinte Banca examinadora:

Orientador: Prof. Dalvan Jair Griebler, Dr.
PUCRS-RS
Faculdade de Sistemas de Informação da SETREM

Orientador: Prof. Samuel Camargo de Souza, M.Sc.
Faculdade de Sistemas de Informação da SETREM

Prof. Vinicius da Silveira Serafim, M.Sc.
Faculdade de Sistemas de Informação da SETREM

Prof. Tiago Seibel, M.Sc.
Faculdade de Sistemas de Informação da SETREM

Profa. Vera Lúcia Lorenset Benedetti, M.Sc.
Coordenação do Curso Bacharelado em Sistemas de Informação
Faculdade de Sistemas de Informação da SETREM

Três de Maio, 19 de junho de 2017.

AGRADECIMENTOS

Ao professor Dalvan Griebler orientador desse trabalho, por aceitar o desafio e nos incentivar.

Ao professor Samuel C. de Souza, pelo apoio dado em todas as horas e por se engajar conosco.

A professora Vera L. B. Lorenset, por acreditar em nosso potencial e na proposta desta pesquisa.

Ao pesquisador do LARCC Carlos Alberto Franco Maron, pelo suporte prestado em todas as horas garantindo assim a continuidade desta pesquisa.

Aos nossos familiares e companheiros pela compreensão e paciência.

RESUMO

Os bancos de dados NoSQL surgem para suprir limitações de bancos de dados relacionais. Com o aumento na sua popularidade há uma expansão no número de sistemas disponíveis, o que dificulta a tomada de decisão quanto à opção que melhor supre as necessidades organizacionais. O objetivo do trabalho é realizar um estudo comparativo de bancos de dados NoSQL de quatro categorias: chave-valor, família de colunas, orientados a documentos e bancos de grafos ou triplos. As abordagens utilizadas foram dedutiva e quali-quantitativa, pois se parte de um ente abstrato para resultar em um estudo bibliográfico de tecnologias e análise estatística de dados. Aplicaram-se técnicas de observação, experimentação e documentação, a fim de examinar e documentar os resultados obtidos. A fundamentação teórica foi elaborada por meio de pesquisa bibliográfica e os resultados obtidos foram um *survey* e análise estatística das ferramentas estudadas. Assim, o trabalho contribuiu para uma análise de diferentes bancos e seu desempenho, com resultados que demonstraram qualitativamente e quantitativamente as características e apontaram as principais vantagens. Ao final foi possível destacar o desempenho dos bancos Couchbase e Aerospike para a carga de trabalho e infraestrutura testadas.

Palavras-chave: NoSQL, banco de dados, análise desempenho.

ABSTRACT

The NoSQL databases were created to overcome the relational databases limitations. With the increase in their popularity there was an expansion in the number of systems available, the decision-making process to decide which option best fits the enterprise needs may be harder because of the amount of options. The goal of this work is to perform a comparative study of 4 categories of NoSQL databases: key-value, column family, document oriented, and graph or triple. The approach included deductive and quali-quantitative tests, because it departs from an abstract entity to create a bibliographical study of technologies, also it was made a statistical analysis of data. It was used the observation technique, experimentations and documentation, to report the results. The theoretical foundation was created based on bibliographical research and the results are demonstrated through a survey and through a statistical analysis of the studied tools. By then, the work contributed to an analysis of different databases and their performance, the results shown qualitatively and quantitatively the characteristics and point out to the main advantages. It also became verified the importance of scientific research, not only to the community but also to the society in general, everyone that uses these technologies. At the end it was possible to highlight the performance of Couchbase and Aerospike databases for the tested workload and infrastructure.

Keyword: NoSQL, database, performance analysis.

LISTA DE QUADROS

LISTA DE FIGURAS

LISTA DE SIGLAS

ACID – *Atomicity, Consistency, Isolation, Durability*

API – *Application Programming Interface*

BASE – *Basic Availability, Soft-state, Eventual consistency*

BSD – *Berkeley Software Distribution*

CAP – *Consistency, Availability, Partition tolerance*

CLI – *Command Line Interface*

CRUD – *Create, Read, Update, Delete*

DBMS - *Database Management System*

DRAM – *Dynamic Random Access Memory*

IMDB – *In-Memory Database*

IMDG – *In-Memory Data Grid*

HDFS – *Hadoop Distributed File System*

HTTP – *HyperText Transfer Protocol*

I/O – *Input/Output*

JMS – *Java Message Service*

JMX – *Java Management Extensions*

JVM – *Java Virtual Machine*

LARCC – *Laboratory of Advanced Researches for Cloud Computing*

LDAP – *Lightweight Directory Access Protocol*

LRU – *Least Recently Used*

MGL – *Multiple Granularity Locking*

MOM – *Message Oriented Middleware*

MQTT – *MQ Telemetry Transport*

NFS – *Network File System*

NoSQL – *Not only SQL*

OLTP – *OnLine Transaction Processing*

OS – *Operating System*

REST – *Representational State Transfer*

SSB – *Star Schema Benchmark*

SETREM – Sociedade Educacional Três de Maio

SGBD – Sistema de Gerenciamento de Banco de Dados

SO – Sistema Operacional

SRAM – *Static Random Access Memory*

SSD – *Solid State Disk*

SSL – *Secure Socket Layer*

STOMP – *Simple Text Oriented Messaging Protocol*

THP – *Transparent Huge Pages*

TPC – *Transaction Processing Performance Council*

YCSB – *Yahoo! Cloud Serving Benchmark*

SUMÁRIO

INTRODUÇÃO

O aumento na disponibilidade do acesso à Internet, bem como a velocidade deste acesso, veio a popularizar ferramentas tecnológicas baseadas em *cloud computing*, o que delineia um novo panorama na oferta de soluções tecnológicas em geral. Com a abstração trazida pelo *cloud computing* e por modelos como SaaS (*Software as a Service*) e IaaS (*Infrastructure as a Service*), para os usuários cada vez mais as soluções tecnológicas correspondem a uma “caixa preta” que devem efetivamente suportar seus processos de negócio.

Para se manterem competitivas e concorrer com os grandes *players* do mercado em que atuam, as pequenas e médias empresas devem oferecer soluções igualmente robustas e confiáveis, com uma fração do orçamento de seus concorrentes, além de preocupar-se com a escalabilidade da solução para quando ocorrer o crescimento da demanda. Neste contexto surgem os bancos de dados NoSQL, sistemas de armazenamento que se propõem a realizar muito mais do que apenas armazenar informações, mas também complementam as soluções tecnológicas oferecendo velocidade e suporte a grandes volumes de dados.

Com o aumento da demanda por este tipo de software, porém, surgiram muitos concorrentes que aparentam homogeneidade em uma avaliação superficial e atendem as necessidades empresariais em maior ou menor grau. A grande quantidade de opções e a falta de estudos comparativos entre estas opções, bem como a pressão mercadológica por uma decisão rápida, leva as empresas à uma decisão sem embasamento, o que ameaça toda a operação de implementação e implantação.

No primeiro capítulo do trabalho está apresentada a contextualização da pesquisa, que delimita o escopo do estudo, bem como apresenta seus objetivos, justificativa, hipóteses, problema que se propõe a solucionar e a metodologia que foi utilizada.

O segundo capítulo, por sua vez, apresenta o referencial teórico que embasa a etapa inicial do trabalho e o projeto como um todo. Além disso neste capítulo está documentado o estudo do estado da arte de *message brokers* e bancos de dados NoSQL.

Já o capítulo três apresenta os resultados obtidos na pesquisa, ou seja, um comparativo das funcionalidades dos bancos de dados NoSQL apresentados no referencial teórico e também consta os resultados dos testes de desempenho realizados.

CAPÍTULO 1: CONTEXTUALIZAÇÃO DA PESQUISA

1.1 TEMA

Análise e avaliação comparativa do desempenho de bancos de dados NoSQL.

1.1.1 Delimitação do Tema

Estudo do estado da arte, mapeamento das opções de ferramentas e triagem das tecnologias de bancos de dados NoSQL (por exemplo, Redis, Memcached, MongoDB). Após definir as tecnologias que se destacaram nas propriedades entendidas como mais relevantes pelo meio acadêmico, realizar uma avaliação comparativa de desempenho com a geração de cargas de trabalho (*workload*) através da aplicação de *benchmarks*, os quais serão estudados e escolhidos de acordo com o suporte aos *softwares* selecionados.

Alguns dos indicadores de desempenho que serão avaliados em relação ao *hardware* são: alocação de memória RAM, alocação de espaço em disco, uso de CPU e taxa de fragmentação da memória; em relação aos bancos de dados NoSQL: *Throughput* de operações, latência de gravação e latência de leitura. A aplicação dos *benchmarks* selecionados foi executada em um ambiente disponibilizado pelo LARCC[1] (*Laboratory of Advanced Researches for Cloud Computing*), e o mesmo foi composto de uma única máquina servidora. Foi avaliado o desempenho das diferentes tecnologias sob cargas de trabalho compostas de leituras e escritas. Mais cenários de avaliação serão elaborados conforme as possibilidades dos *benchmarks* selecionados para avaliação.

[1] LARCC – Laboratório de pesquisas avançadas para computação em nuvem. Website: http://www.larcc.com.br/

O projeto foi realizado durante o período de outubro de 2016 até agosto de 2017 pelos acadêmicos Dinei André Rockenbach e Nadine Anderle como trabalho de conclusão do curso de Sistemas de Informação da Sociedade Educacional Três de Maio – SETREM, na área de Análise de Sistemas, com foco na área de Computação de Alto Desempenho.

1.2 OBJETIVOS

1.2.1 Objetivo Geral

A presente pesquisa busca avaliar e analisar o desempenho de tecnologias de bancos de dados NoSQL.

1.2.2 Objetivos Específicos

- Estudar o estado da arte dos bancos de dados NoSQL.

- Analisar funcionalidades dos bancos de dados NoSQL.

- Pesquisar *benchmarks* aplicáveis a todos os bancos de dados NoSQL.

- Preparar um ambiente para aplicação dos *benchmarks* nas ferramentas avaliadas.

- Aplicar *benchmarks* nos bancos de dados NoSQL.

- Estabelecer a correlação entre as métricas ou indicadores de desempenho.

- Escrever e publicar artigo referente a pesquisa.

1.3 JUSTIFICATIVA

Atualmente existe uma gama expressiva de *softwares*, aplicados em todas as áreas da sociedade, que criam expectativas e necessidades de desempenho excepcional em vários cenários. Dentre os cenários e argumentos para essa necessidade é possível citar impactos em modelos de negócios, custos e gestão empresarial, entre outros.

Para atender tais necessidades, surgiram tecnologias que se propõem a complementar as aplicações desenvolvidas, colaborando com velocidade, capacidade

de processamento e de armazenamento de dados. Dentre essas tecnologias, é possível citar os bancos de dados NoSQL, que permitem às empresas melhorarem seus aplicativos enquanto mantêm o foco nos mercados em que atuam.

Os bancos de dados NoSQL podem ser classificados como aqueles bancos que não requerem um rigoroso esquema para os registros, que possam ser utilizados de forma distribuída em *hardware* comum, e que não utilizem o modelo matemático dos bancos de dados relacionais (FOWLER, 2015). Essa classificação de bancos de dados tem como por objetivo, atender necessidades de velocidade e escalabilidade de modo mais simples do que os bancos de dados relacionais.

Os *benchmarks*, por sua vez, são testes aplicáveis a várias tecnologias, seja de *hardware* ou *software*, a fim de avaliar seu desempenho e compará-los de modo equivalente, obtendo um ponto de referência a partir do qual pode-se classificar os objetos avaliados (OXFORD, 2016). Eles possuem uma sequência de tarefas que o objeto de análise deve executar. A intenção é induzir cada componente até os seus limites para, posteriormente, obter uma média de desempenho. Após, isso tudo é mensurado e documentado, permitindo checar a classificação obtida.

No entanto, o dia-a-dia conturbado de empresas focadas em desenvolvimento de *software* não oferece oportunidades para definir as ferramentas que se adaptam ao seu negócio e auxiliem no atendimento de suas demandas. Isto é agravado pelo fato de que o estudo comparativo de ferramentas requer um investimento considerável de tempo e recursos. Sendo assim, muitas dessas empresas acabam enfrentando os problemas que estas tecnologias se propõem a resolver, sem saber da existência dessas soluções, tais como gerenciamento de *cache* inteligente e alto desempenho na busca de pequenos pacotes de dados estruturados (CARLSON, 2013) e armazenamento de grandes massas de dados sem comprometer desempenho de leitura e escrita.

É preciso considerar também, que uma parcela considerável das empresas que possuem essa realidade são organizações de pequeno a médio porte, as quais possuem algumas limitações quando a variável se trata de investimentos em infraestrutura. Tendo em vista esse cenário, e com o objetivo de melhor definir o escopo da pesquisa optou-se em delimitar o ambiente de aplicação dos testes a uma única máquina servidora.

1.4 PROBLEMA

Dentre os bancos de dados NoSQL estudados, quais oferecem melhor desempenho no quesito velocidade no processamento de dados?

1.5 HIPÓTESES

- O desempenho dos bancos de dados chave-valor é estatisticamente diferente entre as tecnologias avaliadas.

- O desempenho dos bancos de dados família de colunas é estatisticamente diferente entre as tecnologias avaliadas.

- O desempenho dos bancos de dados orientados a documentos é estatisticamente diferente entre as tecnologias avaliadas.

- O desempenho dos bancos de dados grafos é estatisticamente diferente entre as tecnologias avaliadas.

1.6 VARIÁVEIS

- *Benchmarks*

- Métricas de desempenho

- Ferramentas de banco de dados NoSQL

1.7 METODOLOGIA

1.7.1 Abordagem

Conforme apresentado por Lovato (2013), a metodologia de uma pesquisa possibilita aos pesquisadores duas alternativas a serem empregadas. Sendo que a primeira aponta o estudo para o sentido de reunir resultados através do raciocínio indutivo, dedutivo, ou ainda do conjunto hipotético-dedutivo. Por outro lado, o autor também descreve possibilidades qualitativa, quantitativa ou quali-quantitativa.

O método de pesquisa definido foi o dedutivo, pois parte do conhecimento teórico, ou seja, um ente abstrato para algo concreto, obtido através de observações

e experimentos no mundo real. A dedução foi complementada pelo método quali-quantitativo pois os dados obtidos são numéricos, resultados de uma análise estatística e também descritivos em função do *survey* das tecnologias. Os métodos quantitativos utilizados foram definidos com base nos resultados obtidos do *benchmark*, foram aplicados teste de hipóteses (Teste-T e Wilcoxon), teste de normalidade e comparações de médias.

Deste modo as quatro hipóteses levantadas foram validadas através da aplicação de *benchmarks*, que determinam o desempenho da velocidade no processamento de dados das tecnologias de bancos de dados NoSQL. Posterior à aplicação dos testes com o modelo definido foi preciso classificar e analisar os dados coletados.

1.7.2 Procedimentos

Quanto aos métodos de procedimento, o presente trabalho utiliza-se da pesquisa bibliográfica, aplicação de *benchmarks* para coleta de dados e pós coleta compilação e análise dos dados.

A pesquisa bibliográfica, de acordo com a obra de Macedo (1994), pode ser descrita como a busca de informações literárias, utilizando fontes que estão relacionadas ao problema de estudo, podendo estas fontes serem *web sites*, revistas, artigos, livros, enciclopédias, entre outros. Também será realizado um mapeamento sistemático das tecnologias em bases de dados a serem definidas a partir de *strings* de busca (ou *search strings*).

O trabalho de conclusão de curso utilizou o procedimento de pesquisa bibliográfica, com o objetivo de adquirir o embasamento necessário para a busca do preenchimento da lacuna encontrada no problema.

A pesquisa ainda empregou o procedimento experimental pois esse permite operar e modificar as variáveis independentes e ainda medir as variáveis dependentes (LOVATO, 2013). Ainda quando a pesquisa possui abordagem quantitativa a coleta de dados é feita com base em uma trajetória, que pode ser subdividida em: Planejamento, Execução, Coleta de Dados e Análise e Interpretação dos Resultados (STORCK , GARCIA, *et al.*, 2006).

Para validar as hipóteses apresentadas no item 1.5, serão comparadas as métricas obtidas da aplicação dos *benchmarks* nos bancos de dados e será estabelecido se existe diferença entre as distintas tecnologias.

1.7.3 Técnicas

Durante a execução deste trabalho de conclusão de curso o mesmo será amparado por algumas técnicas, podendo destacar a documentação e observação. A documentação se encaixa no momento em que se transcreve os resultados obtidos através da observação, a fim de resguardá-los. A observação será empregada durante a execução dos *benchmarks* para realizar a obtenção dos indicadores.

Além disso foi aplicada a técnica experimental, simulando um ambiente isolado onde as ferramentas foram instaladas, e submetidas as cargas induzidas de trabalho, com o propósito de classificá-las quanto a seu desempenho quando submetidos a determinadas operações, tais como leitura de gravação.

CAPÍTULO 2: REFERENCIAL TEÓRICO

Na realização de um estudo é necessário buscar o conhecimento das áreas envolvidas, para isso é necessário executar uma revisão literária a fim de aclarar as ações que são realizadas. O objetivo da fundamentação teórica é justificar e explicar o que está sendo desenvolvido.

2.1 DEFINIÇÃO DE TERMOS

2.1.1 XML

O XML é a sigla para *eXtensible Markup Language* (linguagem de marcação extensível, em tradução livre) e sua sintaxe oferece mecanismos para descrever documentos estruturados (SALMINEN e TOMPA, 2012). Ele foi desenvolvido pela W3C (World Wide Web Consortium) com o objetivo de superar algumas limitações que o HTML possuía (MARCHAL, 2002).

Ele é largamente utilizado na tecnologia da informação para transferência e armazenamento de informações, inclusive sendo a estrutura que alguns IMBDs utilizam para armazenar seus dados. Um exemplo disso é o XML-IMDB desenvolvido pela empresa QuiLogic (MARQUES, 2002).

2.1.2 JSON

JSON (*JavaScript Object Notation*, ou Notação de Objetos JavaScript em tradução livre) é um formato de troca de dados de fácil leitura e escrita para humanos e de fácil análise e geração para máquinas, leve e independente de linguagem de programação, que tem suas origens na linguagem de programação JavaScript (CROCKFORD, 2006).

No ano de 2001, Crockford documentou a utilização do JSON em uma página da Web e na sequência a sua utilização se alastrou e passou a ser uma alternativa para algumas limitações que o XML traz (SMITH, 2015). Com a popularização e as vantagens perante aos padrões anteriores, o JSON foi adotado por vários bancos de dados como padrão de armazenamento, como por exemplo o MongoDB.

2.1.3 Avaliação de Desempenho de Software

A avaliação de desempenho de softwares é feita através da coleta de indicadores ou métricas de desempenho, porém, como destacam Woodside, Franks e Petriu (2007), o desempenho de softwares é uma qualidade de difícil entendimento, porque ela é afetada por todos os aspectos do projeto, codificação e ambiente de execução.

Tendo em vista a complexidade da avaliação de desempenho de softwares, a coleta das métricas de desempenho geralmente se dá através da aplicação de *benchmarks* ao produto de software em questão (MARON, 2014), onde busca-se isolar alguns aspectos que possam influenciar no desempenho (tais como o ambiente de execução).

2.1.4 *Benchmark*

O objetivo geral da aplicação de um *benchmark,* de acordo com a literatura, é a identificação de atividades que possibilitam alcançar desempenhos superiores. Os *benchmarks* são compostos por um sistema estruturado, onde é realizada a análise das etapas uma a uma, assim sendo possível identificar o modo de trabalho. (MORAES, 2012). Essa metodologia não é utilizada apenas na área de tecnologia da informação, mas também em áreas como gestão empresarial, gestão de pessoas e processos de produção, entre outros.

Na tecnologia da informação o *benchmark* pode ser conceituado como uma sequência de medições dos resultados gerados, por exemplo, por um sistema ou uma rede. Tais resultados são comparados com outras tecnologias similares e com base nesta comparação é possível identificar melhorias a serem realizadas. Para que um *benchmark* possa ser classificado de qualidade é preciso que atenda algumas características: escalável, portável, relevante e compreensível (MANNINO, 2008).

Huppler (2009) defende que todo bom benchmark possui cinco características principais, sendo: relevante, pois leitores dos resultados devem crer que eles trazem informações importantes; repetível, onde há confiança de que pode-se repetir os testes e obter o mesmo resultado; justo, já que todos os elementos comparados podem participar de forma igualitária; verificável, pois há confiança de que os resultados documentados são reais; e econômico, para que os patrocinadores possam arcar com os custos de rodar os testes .

2.1.5 Cargas de trabalho

As cargas de trabalho ou *workloads* são conjuntos de instruções que são enviadas pelos *benchmarks* aos sistemas a serem avaliados, de forma que seja possível realizar uma comparação do desempenho destes. Durante a geração destas cargas e o processamento das mesmas pelos sistemas, é realizado o monitoramento do comportamento dos mesmos a fim de obterem-se informações sobre o desempenho dos mesmos.

Cada carga de trabalho representa um conjunto particular de operações de leitura e/ou escrita, tamanho de dados, distribuição de requisições, etc, e pode ser utilizada para avaliar sistemas em um ponto definido no espectro do desempenho (COOPER, SILBERSTEIN, *et al.*, 2010).

2.1.6 Tempo de execução de tarefas

O tempo de execução de uma tarefa pode ser interpretado de dois pontos de vista: o do usuário e o do projetista de *software* ou *hardware*. Para o usuário a informação relevante é o tempo de relógio, enquanto que para o projetista é a velocidade de execução, que está relacionada com o tempo de execução na CPU e/ou com o tempo de entrada e saída. O tempo de execução do CPU é tempo gasto computando a tarefa, e não inclui o tempo que está sendo aguardado as entradas e saídas. (HENNESSY e PATTERSON, 2014)

2.1.7 Memória Principal

O sistema de memória de um computador é tão importante quanto seu processador, se levado em consideração os fatores desempenho e usabilidade. As memorias de um computador podem ser classificadas como: Memória Principal, Memória de *Cache* e Memória de Massa. (PARHAMI , 2007)

Existem vários tipos de memória RAM, elas podem ser adjetivadas como voláteis, pois perdem seus dados quando não existe fonte externa de energia. Os dois principais tipos citados pela literatura são SRAM e DRAM. O primeiro é basicamente um grande vetor de células de armazenamento, as quais são acessadas como registradores e os dados permanecem em memória enquanto existir fonte externa de energia. Já as *Dynamic Random Acess Memory* devem restaurar os dados armazenados periodicamente evitar perdas. (PARHAMI , 2007)

2.1.8 ACID

A ACID (*Atomicity, Consistency, Isolation, Durability*) é um modelo de consistência que se baseia em quatro pilares para oferecer a garantia de que uma vez que os dados estão gravados, tem-se consistência completa nas leituras (ABRAMOVA, BERNARDINO e FURTADO, 2014). Este modelo de consistência é o extremamente comum em bancos de dados relacionais, e pouco adotado em bancos NoSQL (ainda que o suporte a ele tenha crescido nos últimos anos).

A *Atomicity* ou Atomicidade estipula que cada operação deve afetar apenas os dados especificados, e nenhum outro, enquanto que a *Consistency* ou Consistência garante que todas as operações devem manter a base de dados em um estado consistente, ainda que sejam interrompidas (FOWLER, 2015).

Isolation ou Isolamento, como seu nome sugere, defende que operações concorrentes não podem afetar umas às outras, e a *Durability* ou Durabilidade oferece garantias de que os dados gravados não serão perdidos, uma vez que a transação de escrita tenha retornado sucesso na operação (FOWLER, 2015).

2.1.9 BASE

O modelo de consistência BASE (*Basically Avaliable, Soft state, Eventually consistent*), em oposição ao modelo ACID, oferece consistência eventual, ou seja, uma vez que os dados estão gravados, eventualmente estes estarão disponíveis para a leitura (FOWLER, 2015).

O termo *Basically Avaliable* ou Basicamente Disponível implica que o retorno da base de dados a uma consulta pode ser uma mensagem de falha ao obter os dados ou eles podem vir em um estado inconsistente, *Soft state* ou Estado flexível representa

a não-obrigatoriedade do banco de dados de prover um estado consistente durante a sua operação, havendo a possibilidade de um estado de “transição” (VOHRA, 2015).

Por final, o termo *Eventually consistent* ou Eventualmente consistente provêm da possibilidade do banco de dados de receber mais dados mesmo que os dados já presentes na base não estejam em um estado perfeitamente consistente (VOHRA, 2015). A consistência eventual pode ser classificada de acordo com as garantias oferecidas pela implementação do sistema de banco de dados: consistência casual, consistência “leia-suas-gravações”, consistência na sessão, consistência de leitura monotônica e consistência de escrita monotônica (SULLIVAN, 2015).

2.1.10 OLTP

Os sistemas OLTP (*OnLine Transaction Processing*) são aplicações voltadas à transações online, ainda que o termo “transações” seja ambíguo entre transações de bancos de dados e transações de negócio ou comerciais. Sistemas OLTP possuem foco em disponibilidade, velocidade, concorrência e resiliência.

Algumas das principais características de ambientes OLTP são: tempo de resposta curto, transações pequenas, operações de manutenção de dados, grandes populações de usuários, alta concorrência, altos volumes de dados, alta disponibilidade e uso de dados relacionados a ciclos temporais (ORACLE, 2011).

2.1.11 Desnormalização

Com o crescimento dos bancos de dados relacionais e seu foco na consistência dos dados, ganhou força a ideia da normalização dos dados, onde busca-se garantir que as informações não estejam armazenadas de forma redundante no banco de dados para evitar anomalias nas operações com os mesmos.

Contudo, na maioria dos modelos de dados a normalização afeta o desempenho da consulta dos dados, devido ao fato de a informação não estar armazenada da forma como é utilizada pela aplicação. Muitos bancos NoSQL, principalmente os de modelo *columnar*, defendem a desnormalização dos dados, ou seja, o não-seguimento das regras das formas normais e o consequente armazenamento redundante de informações de forma a melhorar a velocidade das consultas aos dados.

2.1.12 Teorema CAP

O teorema CAP (*Consistency, Availability* e *Partition tolerance*) foi proposto por Eric Brewer em (BREWER, 2000) e verificado por Gilbert e Lynch (2002), desde então passou a ser largamente aceito pela academia.

No teorema CAP, *Consistency* (consistência) é sinônimo de linearizabilidade (*linearizability*), um termo introduzido em (HERLIHY e WING, 1990). Simplificadamente, um sistema é considerado consistente pelo teorema CAP se as operações distribuídas sejam vistas como se estivessem executando em um único nó. Ou seja, se a operação B começou após a operação A terminar, a operação B deve enxergar o sistema no mesmo estado em que ele estava após a completude de A, ou em um estado mais novo (KLEPPMANN, 2015). Também vale mencionar que esta definição difere da Consistência como definida na sigla ACID.

A definição de *Availability* (disponibilidade) impõe que cada requisição recebida por um nó que não esteja em estado de falha deve resultar em uma resposta (que não seja um erro) (GILBERT e LYNCH, 2002). Kleppmann (2015) ressalta que, com esta definição, qualquer nó do cluster deve ser capaz de responder a requisição de forma independente de outros nós.

Por fim, *Partition tolerance* (tolerância a partições) afirma que redes assíncronas podem perder pacotes, e que uma rede está particionada quando todas as mensagens entre dois componentes da rede são perdidas. Kleppmann (2015) critica que esta definição ignora latência e tempos de resposta, então um nó pode demorar um tempo arbitrário para enviar uma resposta e ainda assim ser considerado disponível.

O teorema afirma que na existência de uma falha de comunicação (*partition*) cada nó de um sistema distribuído deve escolher entre responder requisições, mantendo a disponibilidade (*availability*) e assumindo o risco de não retornar os dados mais atuais, ou rejeitar requisições para garantir a consistência dos dados (*consistency*). Sistemas classificados como AP priorizam a disponibilidade, enquanto que sistemas classificados como CP priorizam a consistência.

O teorema tem sido alvo de muitas críticas, e Brewer explora algumas de suas limitações em (BREWER, 2000), enquanto que Abadi propõe o teorema PACELC como alternativa em (ABADI, 2012).

2.2 MÉTRICAS DE DESEMPENHO

As métricas ou medidas de desempenho são valores obtidos ou calculados a partir das informações coletadas durante a execução de uma carga de trabalho (KANG, JIN, *et al.*, 2014), geralmente gerada pela execução de um *benchmark*.

2.2.1 Tempo de uso de CPU

Geralmente se distingue o desempenho do CPU, com base no tempo que o mesmo está trabalhando a favor das tarefas. Existem três tipos de classificações de tempo de execução de CPU. O primeiro é denominado tempo de execução de CPU ou simplesmente tempo de CPU, ele define o quanto a CPU precisa para calcular uma tarefa específica. Já o tempo de CPU do usuário é o tempo gasto por um programa literalmente. E por fim existe o tempo de CPU do sistema que pode ser definido como aquele tempo destinado as tarefas do sistema operacional em favor do programa. (PATTERSON e HENNESSY, 2014)

2.2.2 Taxa de fragmentação da memória

Esta métrica reflete o nível de fragmentação da memória, sendo definida como o resultado da divisão entre a memória utilizada e a memória física alocada pelo sistema operacional (o *Resident Set Size*, ou RSS) (CAO, SAHIN, *et al.*, 2016).

2.2.3 *Throughput* (ops/sec ou msg/sec)

No caso dos bancos de dados em memória esta métrica mede a quantidade de operações realizadas por segundo, enquanto que para os *message brokers* ela corresponde à quantidade de mensagens recebidas pelo consumidor ou enviadas pelo produtor por segundo.

Esta métrica detalha o número de operações por segundo para uma carga de trabalho, refletindo a eficiência do processamento de requisições (CAO, SAHIN, *et al.*, 2016).

2.2.4 Latência de gravação/leitura de dados

Em sistemas distribuídos, o termo “latência” pode ser utilizado para se referir ao tempo de processamento de várias etapas dos processos, porém o presente trabalho possui foco no tempo decorrido entre o envio de um comando pelo cliente até

o recebimento da resposta do comando, o que é conhecido como *round-trip latency* (NELSON, 2016).

A latência de gravação de dados, neste contexto, é definida como o tempo decorrido entre o envio de um comando que tem como finalidade gravar dados no banco de dados, e a resposta do sistema de que os dados foram armazenados.

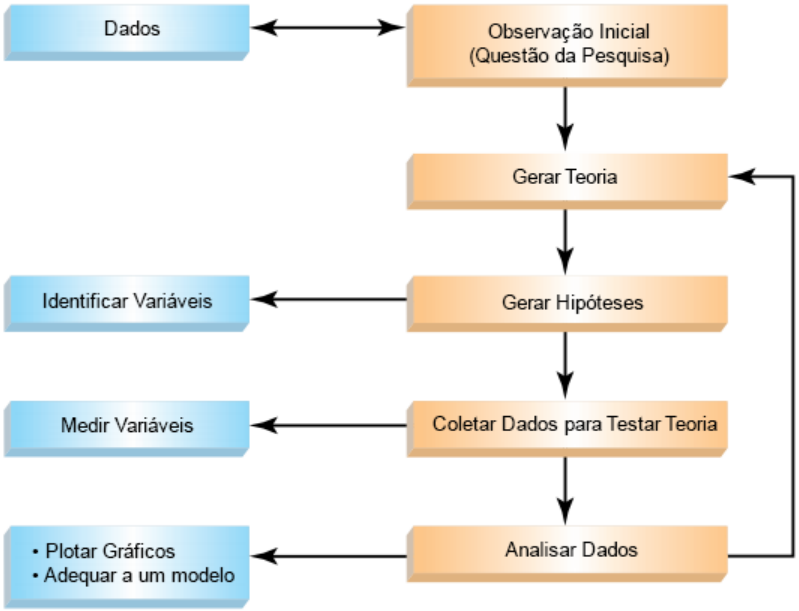
A latência de leitura de dados é o tempo decorrido entre o envio de um comando por parte do cliente, que tenha como finalidade obter dados do banco de dados e o recebimento dos dados esperados do sistema.

2.3 ANÁLISE ESTATÍSTICA

A sociedade utiliza a ciência quando tem a necessidade de descobrir algo novo ou explicar algum fenômeno. Para quaisquer destes fenômenos que se pretende explicar, devem ser coletados dados, e então obter conclusões a partir da sua análise (FIELD, 2009).

O processo de pesquisa é um conjunto de etapas que devem ser seguidas para se responder a uma pergunta. Apesar de sua complexidade a Figura 1 busca sintetizar este processo.

Figura 1 - Processo de Pesquisa



Fonte: Adaptado de (FIELD, 2009)

Para iniciar a observação é preciso gerar possíveis explanações, ou teorias, e a partir destas observações é possível realizar algumas constatações, que são denominadas hipóteses. Após esses passos, é necessário realizar a coleta dos dados, que permitem testar as previsões levantadas. Com a análise destes dados é possível realizar a comprovação da teoria levantada ou ainda assim modificar tal teoria caso seja constatado que isso é necessário (FIELD, 2009).

Com a geração das hipóteses começam a se definir as variáveis, que de acordo com a literatura são características que podem atribuir diferentes valores, dentre exemplos, pode ser utilizado idade, peso, rendimento salarial entre outros (BISQUERRA, SARRIERA e MATÍNEZ, 2007). Após essa etapa, no momento em que é realizada a coleta de dados e testes da teoria, é possível empregar a medição das variáveis e por fim compilar os dados.

2.3.1 Teste de hipótese estatística

Um teste de hipótese é um processo da estatística que utiliza amostras para testar uma afirmação referente a um valor de um parâmetro populacional. De acordo com a literatura de Larson e Farber (2010), áreas como psicologia, medicina e negócios confiam nesse tipo de testes para realizarem tomadas de decisões.

O teste de hipótese, por exemplo, pode ser usado para mensurar o consumo de combustível de determinado modelo de veículo. Não há como medir o consumo de todos os automóveis, porém, se escolhida uma amostra relevante e aleatória é possível presumir, que os outros carros terão o comportamento similar (LARSON e FARBER, 2010).

Para compreender o teste de hipótese estatística, é preciso entender como é realizada a construção destas afirmações. De acordo com Larson e Farber (2010) o pesquisador deve afirmar com cautela um par de hipóteses. Uma delas será a afirmação e por consequência, a outra, o complemento. É preciso considerar que quando uma delas for falsa a outra será verdadeira e vice-versa.

Esse tipo de par de hipóteses pode ser classificado como hipótese nula e hipótese alternativa ou experimental, porém qualquer uma delas pode ser a representação da afirmação original. O que é preciso ter claro é que caso a primeira afirme que existe uma correlação entre as variáveis, por consequência a hipótese nula

precisa afirmar o contrário, ou seja, que não há relação dentre as variáveis (DANCEY, 2006).

2.3.2 Nível de significância estatística e tipos de erro

Em um teste de hipótese, o nível de significância pode ser conceituado como a probabilidade máxima admitida para cometer um erro do tipo I, o qual é representado pela letra grega alfa α. Quando se parametriza o nível de significância com um valor pequeno, isto indica que há uma procura por uma probabilidade pequena de rejeição da hipótese nula. De acordo com a literatura existem três níveis geralmente utilizados, são eles: α=0,10, α=0,05 e α=0,01 (FERREIRA, 2005).

A probabilidade de cometer um erro do tipo II é definida pela letra grega beta β. Porém, a determinação desse valor é mais difícil, pois habitualmente não são atribuídos valores fixos para o parâmetro da situação alternativa. É possível atribuir valores escolhidos no caso alternativo, e encontrar os valores que correspondem a β (GUIMARÃES, 2010).

2.3.3 Métodos estatísticos

Os testes de hipóteses são classificados em dois grupos, paramétricos e não paramétricos. Os testes paramétricos podem ser aplicados quando a variável possui distribuição de probabilidades teórica conhecidas. Além disso, é suposto que a variável passou por medida no mínimo em nível intervalar, e em alguns casos é preciso que as variáveis aleatórias que estão envolvidas tenham variância homogênea (LOESCH, 2012).

Os testes não paramétricos por sua vez, são aplicados em situações em que a distribuição da população é desconhecida, ou ainda quando se realiza testes sobre o modo de distribuição. São mais genéricos, e podem ser aplicados a variáveis ordinais. Contudo, de acordo com a literatura, não são tão potentes quanto os paramétricos devido à falta de informações (LOESCH, 2012).

2.3.3.1 Teste de Lilliefors

Este teste é uma adaptação *Kolmogorov-Smirnov* utilizado para testar a normalidade de uma amostra.

2.3.3.2 Test-T

O test-T ou test-t de Student, é um teste estatístico empregado para definir se o valor médio de uma variável ininterrupta, difere significativamente daquele em outra amostra. Em sua obra Hulley, Cummings, *et al* (2015), citam como exemplo, um estudo onde seus participantes foram tratados com dois medicamentos diferentes, um teste t nesse cenário pode ser utilizado para comparar os escores médicos após o tratamento nos dois grupos (teste t para duas amostras não pareadas), ou ainda uma mudança média que parte da linha de base até após o tratamento em ambos grupos (teste t para duas amostras pareadas) (HULLEY, CUMMINGS, *et al.*, 2015).

2.3.3.3 ANOVA

O teste paramétrico que é empregado para três ou mais condições é denominado Análise de Variância (ANOVA). Há dois tipos de ANOVA uma que atende a grupos independentes e outra para o refinamento de medidas repetidas. Já para os testes não paramétricos, é possível aplicar avaliações equivalentes, para grupos independentes a ANOVA de Kruskal-Wallis e a ANOVA de Friedman atende as medidas repetidas (ROWE, REIDY e DANCEY, 2017).

A ANOVA pode ser considerada uma expansão do teste t. Mesmo que uma ANOVA seja aplicada em dois grupos, substituindo assim teste t, os resultados obtidos devem ser os mesmos, ainda que para a ANOVA a estatística de teste seja nomeada de F (ROWE, REIDY e DANCEY, 2017).

2.3.3.4 Teste Z

O teste Z é usado para realizar a comparação de proporções para determinar se essas são diferentes uma da outra de modo estatisticamente significativo. Este tipo de teste pode ser usado para hipóteses unilaterais e bilaterais. De acordo com um exemplo de Hulley, Cummings, *et al* (2015), um teste Z unilateral permite determinar se a amostra de presidiários com diabetes, é significativamente maior que a amostra de pessoas que estão fora do sistema penitenciário e que possuem diabete. Além disso, um teste Z bilateral poderia ser empregado para determinar se a proporção de presidiários com diabetes é significativamente diferente (i.é, menor ou maior) do que a amostra de pessoas que estão fora do sistema penitenciário com diabetes (HULLEY, CUMMINGS, *et al.*, 2015).

2.3.3.5 Teste de hipóteses múltiplas

Esse teste é indicado para cenários onde o investigador examina mais de uma hipótese de estudo, aumentando, assim, o risco de incorrer em um erro Tipo I, a menos que o nível de significância estatística seja ajustado. Um bom exemplo de aplicação dessa técnica é, mesmo que o pesquisador tenha relatado uma associação estatisticamente significativa (P=0,03) dentre a ingestão de vitamina D e declínio cognitivo, os seus resultados foram questionados, pois o mesmo não considerou o efeito do teste de hipóteses múltiplas, porque a pesquisa havia examinado 30 suplementos nutricionais (HULLEY, CUMMINGS, *et al.*, 2015).

2.3.3.6 Teste dos sinais

Essa técnica é um teste não paramétrico, e pode ser utilizada para testar uma mediana de população confrontando um valor hipotético *k.* Ele pode ser unilateral à esquerda, ou à direita, ou ainda bilateral. Para aplicar o teste de sinais, inicialmente é preciso comparar as entradas na amostra com a mediana hipotética *k.* Caso a entrada fique abaixo da mediana, então é designado um sinal negativo (-), por outro lado, se a entrada está acima da mediana, é designado um positivo (+), e ainda se a entrada for equivalente à mediana, então é designado um 0 (zero). Então, é realizada uma comparação do número de sinais positivos e negativos, ignorando os 0. Caso haja uma grande diferença entre a quantidade de sinais + e -, então é possível acreditar que a mediana é diferente do valor hipotético, ou seja, a hipótese nula pode ser rejeitada (FARBER, 2010).

2.3.3.7 Teste de Wilcoxon

Este teste pode ser aplicado para determinar se duas amostras dependentes foram escolhidas em populações que tem a mesma distribuição (FARBER, 2010).

O teste de Wilcoxon é uma alternativa ao teste paramétrico teste t para quando os dados não seguem distribuição normal. Ele também é considerado pela literatura uma extensão do teste de sinais, porém, mais potente, isso porque além da direção da diferença de cada par, ele leva em consideração a magnitude da diferença dentro dos pares (FÁVERO e FÁVERO, 2015).

2.3.3.8 Teste de U Mann Whitney

O teste de Mann-Whitney ou U Mann Whitney, é indicado para realizar teste de hipótese nos quais as duas populações possuem igual distribuição. Sendo assim, esse tipo de teste não trabalha com hipóteses sobre os parâmetros (VIEIRA, 2011). Assim como o teste de Wilcoxon, esse teste é uma versão não-paramétrica equivalente ao teste t (*Student*). Para um teste bilateral a hipótese nula é de que a mediana das duas populações é igual (FÁVERO e FÁVERO, 2015).

2.3.3.9 Teste de Kruskal-Wallis

É um teste não paramétrico, que pode ser aplicado para definir se três ou mais amostras independentes foram selecionadas de populações que possuem a mesma distribuição. Para utilização deste tipo de teste é necessário que cada amostra seja selecionada aleatoriamente, e ainda que o tamanho dessas seja no mínimo 5. Caso tais condições sejam atendidas, a distribuição de amostragem para o teste de Kruskal-Wallis é aproximada por uma distribuição qui-quadrado com graus de liberdade k-1 (FARBER, 2010).

2.3.3.10 Teste de qui-quadrado

O teste qui-quadrado é uma técnica estatística, que confronta duas (ou mais) proporções para determinar se elas são estatisticamente diferentes entre si. Por exemplo, uma pesquisa definiu se o risco de demência era similar em indivíduos que praticam exercícios físicos ao menos duas vezes por semana, e em pessoas que faziam exercícios com menor frequência, comparando estatisticamente os riscos por meio do teste do qui-quadrado (HURWITZ, NUNGENT, *et al.*, 2016).

2.4 ESTUDO DE *BENCHMARKS*

Uma vez que *benchmarks* são largamente utilizados para avaliar sistemas computacionais, e que existem *benchmarks* para variados níveis de abstração, desde o CPU até sistemas de bancos de dados e sistemas empresariais completos (COOPER, SILBERSTEIN, *et al.*, 2010), é relevante avaliar os benchmarks disponíveis que sejam adequados à avaliação proposta pelo presente estudo.

2.4.1 Yahoo! Cloud Serving Benchmark (YCSB)

O *Yahoo! Cloud Serving Benchmark* (YCSB) busca oferecer um conjunto de cargas de trabalho para avaliar o desempenho de sistemas de armazenamento de chave-valor e de nuvem (YAHOO, 2010).

Como destacam Cooper, Silberstein, *et al.* (2010), as aplicações práticas dos bancos de dados emergentes diferem muito das cargas de trabalho geradas pelos *benchmarks* tradicionais (tais como o TCP-C), portanto o YCSB busca facilitar comparações de desempenho destes novos sistemas.

2.4.2 TPC-C

O TPC-C é um dos *benchmarks* desenvolvidos e mantidos pelo *Transaction Processing Performance Council* (TPC) e é caracterizado como uma carga de trabalho OLTP. Ele é um conjunto de transações com operações intensivas de leitura e atualização de dados que simulam as atividades encontradas em ambientes complexos de aplicações OLTP (TPC, 2010).

Algumas das principais características do *benchmark* TPC-C são: a execução simultânea de múltiplas transações online e agendadas, múltiplas sessões online, tempo de execução de aplicação e sistema moderado, *input/output* de disco significante, integridade transacional (propriedades ACID) e distribuição não-uniforme de acesso a dados. O objetivo do dos *benchmarks* do TPC é prover aos usuários da indústria dados de desempenho relevantes e objetivos (TPC, 2010).

2.4.3 TPC-H

O TPC H é de acordo com seu desenvolvedor um *benchmark* de apoio à tomada de decisão. Incide em uma série de consultas ad-hoc focadas para negócios e alterações de dados em paralelo. As pesquisas e os dados que alimentam a base de dados, foram selecionados para terem ampla relevância para o setor de indústria (TPC BENCHMARK™, 2017).

Este *benchmark* simula sistemas de suporte à decisão, os quais que analisam grandes massas de dados, disparam consultas complexas e fornecem respostas a questões críticas no âmbito de negócios. A métrica de desempenho proposta pelo TPC-H é denominada de métrica de desempenho de consulta por hora do TPC-H (*QphH @ Size*) e reflete várias características da capacidade do sistema de processar

pesquisas. Essas características abrangem o tamanho de banco de dados versus quais consultas são executadas, a capacidade de processamento de consulta quando essas são enviadas por um único fluxo, e a taxa de transferência da consulta o que se aplica quando as consultas são enviadas por vários usuários simultâneos (TPC BENCHMARK™, 2017).

2.4.4 Star Schema Benchmark

O *Star Schema Benchmark* (SSB) foi desenvolvido com o propósito de medir o desempenho de sistemas de banco de dados em suporte a aplicações clássicas de data *warehousing*, e é baseado no *benchmark* TPC-H (O'NEIL, O'NEIL, *et al.*, 2009).

Essa adaptação do TPC-H, foi realizada através da união das tabelas fatos, denominadas *LineItem* e *Order*, resultando assim na tabela de fatos chamada *LineOrder*, que por sua vez é composta por todos os atributos que pertenciam as tabelas originais. Além disso os autores do *benchmark*, excluíram a tabela de fatos PartSupp. Isso se deu devido a diferenciação da granularidade temporal, tendo em vista que as atualizações aplicadas sobre as tabelas *LineOrder* e *PartSupp* não ocorriam ao mesmo tempo (O'NEIL, O'NEIL, *et al.*, 2009).

Também foi feita a exclusão de determinados atributos das tabelas *LineItem* e *Order* do TPC-H, pois se tratavam de campos *strings* não estruturados, os quais não podem ser agrupados ou sumarizados, e nem aplicados para contextualizar fatos analisados. E por fim, foi acrescentada a tabela de dimensão *Date*, para atender a necessidade de armazenamento do histórico de vendas (O'NEIL, O'NEIL, *et al.*, 2009).

Diferente do TPC-H e do TPC-DS, o SSB tem as suas tabelas de dimensão desmoralizadas, com isso ele ganha mais desempenho na capacidade de processamento de consultas, e também torna-se mais simples e fácil para os usuários compreenderem e manipularem os dados (KIMBALL e ROSS, 2013).

2.4.5 LinkBench

O LinkBench foi desenvolvido pelo Facebook, ele se trata de um *benchmark* que tem por objetivo avaliar o desempenho de bancos de dados para cargas de trabalho, similares às da implantação MySQL (ORACLE MYSQL) de produção da rede social. De acordo com seu desenvolvedor o LinkBench é possui diversas possibilidades de configuração e extensão. Podendo ser reconfigurado, para simular

diversas cargas de trabalho, e os *plugins* podem ser implementados para *benchmarking* de sistemas de banco de dados adicionais. O LinkBench é distribuído sob a Licença Apache, Versão 2.0 (FACEBOOK, 2015).

O Facebook representa grande parte dos seus dados como um gráfico social, com as informações armazenadas em um banco de dados MySQL. O objetivo do LinkBench é emular a carga de trabalho de banco de dados de gráfico social e fornecer um *benchmark* realista para desempenho de banco de dados em cargas de trabalho sociais (FACEBOOK, 2015).

2.4.6 TPC-W

O TPC-W é um *benchmark* que simula uma carga de trabalho de comércio eletrônico, e também as atividades de um *website* de empreendimento de varejo. Os usuários emulados podem navegar e realizar compras a partir do site. No caso de TPC-W os produtos são livros. É simulado um usuário é através de um emulador de navegador remoto, RBE, que gera o mesmo tráfego de rede HTTP que seria recebido por um cliente real utilizando um navegador. Além disso é possível, se conectar um navegador verdadeiro e encomendar livros no site TPC-W (TPC BENCHMARK™, 2017).

2.4.7 BigBench

O BigBench é um benchmark de ponta a ponta baseado em TPC-DS (TPC BENCHMARK™). O TPC-DS foi projetado com um esquema *multiple-snowflake* preenchido com dados estruturados, isso possibilita simulações de todos as abrangências de sistemas comerciais de apoio à tomada de decisão. O esquema do floco de neve é projetado utilizando um exemplo de varejo que incide de três canais de negociação, Armazenamento, Web e Catálogo, complementado ainda por uma tabela de fatos Inventário (BARU, BHANDARKAR, *et al.*, 2014).

O BigBench utiliza os dados dos canais de distribuição de vendas *Store* e *Web* do TPC-DS, e expande adicionando dados semiestruturados e não estruturados. A parte estruturada é proveniente do modelo de dados do TPC-DS. Enquanto isso, a parte semiestruturada armazena, utilizando arquivos de logs, o fluxo de requisições de um usuário na Web, de modo estruturado para compras realizadas, e de modo não estruturado para os outros tipos de navegação. Por fim, a parte não estruturada

contém textos relacionados aos itens e também avaliação de produtos, gerados a partir do modelo de cadeias de Markov (RABL, FRANK, *et al.*, 2014).

No que diz respeito a carga de trabalho, o BigBench possui um total de 30 consultas, as quais cobrem diferentes categorias de análise em Big Data do ponto de vista de negócios (RABL, FRANK, *et al.*, 2014).

2.4.8 BigDataBench

BigDataBench é um conjunto de *benchmark* desenvolvido para *Big Data Computing*. Ele é abstraído de cargas de trabalho de serviço típicas da Internet e engloba conjuntos de dados representativos e aplicativos amplos. O BigDataBench sintetiza seis cenários de aplicação, incluindo três domínios de aplicação importantes: motor de busca, redes sociais e comércio eletrônico. Além disso, ele oferece três cenários básicos de operação: *micro-benchmark*, consulta relacional e operações de armazenamento de dados básicas (LIANG, FENG, *et al.*, 2014).

Esses cenários oferecem dezenove cargas de trabalho típicas de aplicativos/algoritmos, que consideram o serviço on-line, o serviço off-line e a análise em tempo real. Os tipos de dados dessas cargas de trabalho podem ser classificados em dados estruturados, semi-estruturados e não estruturados com formatos de dados de texto, tabela ou gráfico (LIANG, FENG, *et al.*, 2014).

O BigDataBench também dispõe de um gerador de dados para *benchmarks* baseados em conjuntos de dados de cenários reais, e gera os seis modelos base extraindo as características sintéticas de conjuntos de dados correspondentes, incluindo entradas na Wikipédia, avaliações de filmes Amazon, dados de transações de *e-commerce* entre outros. Os usuários podem gerar dados sintéticos escalando os modelos de base, mantendo as características dos dados (LIANG, FENG, *et al.*, 2014).

2.5 NOSQL

Fowler (2015) afirma que podem ser definidos como bancos de dados NoSQL aqueles bancos que não requerem um rigoroso esquema para os registros, que possam ser utilizados de forma distribuída em *hardware* comum, e que não utilizem o modelo matemático dos bancos de dados relacionais.

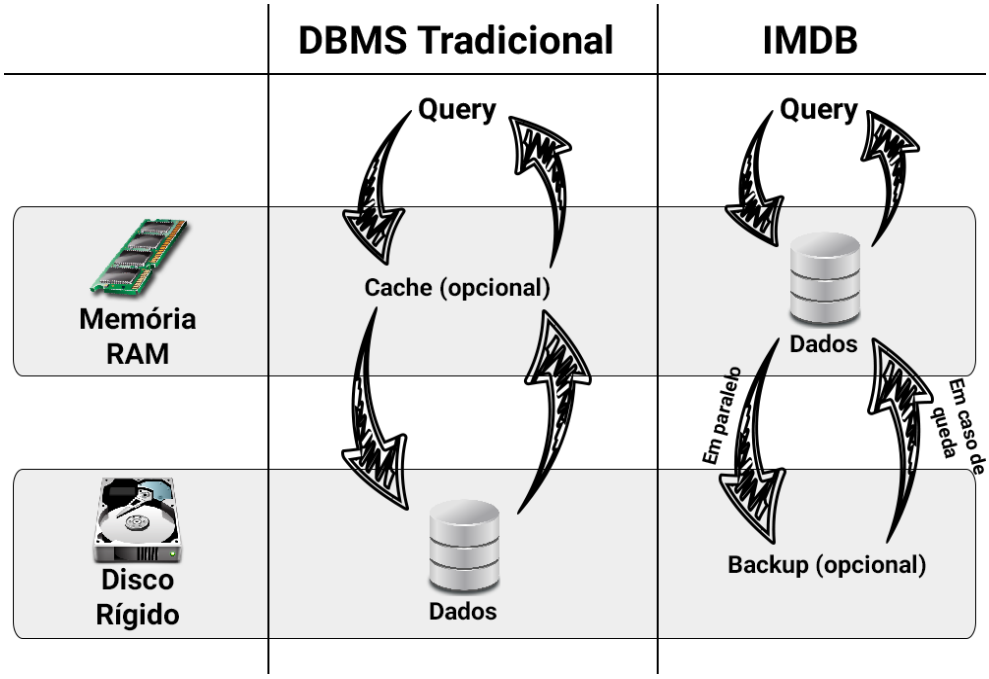
Segundo Kabakus e Kara (2016), bancos de dados relacionais (RDBMS) se baseiam no modelo ACID (*Atomicity*, *Consistency*, *Isolation*, *Durability*) para garantir a consistência e manter a integridade dos dados, enquanto que os bancos NoSQL partem do princípio BASE (*Basically Available, Soft-state, Eventually consistent*) para atingir melhor desempenho, disponibilidade e escalabilidade. Porém, segundo Fowler (2015, p. 18), vários bancos NoSQL têm adicionado suporte ao modelo ACID nos últimos anos.

Os bancos de dados NoSQL podem possuir armazenamento primário tanto na memória RAM quanto o modelo tradicional em discos. Ambos modelos de armazenamento possuem suas vantagens e desvantagens, e podem ser utilizados em complemento um ao outro.

Os bancos de dados em memória ou *in-memory databases* (IMDB), como seu próprio nome sugere, são bancos de dados que utilizam a memória principal do computador ou memória RAM como dispositivo principal de armazenamento de dados (LAKE e CROWTHER, 2013, p. 183). O desempenho destes sistemas de armazenamento é significantemente melhor do que sistemas tradicionais devido ao uso da memória volátil para o mapeamento de registros (ABRAMOVA, BERNARDINO e FURTADO, 2014).

O principal motivo para o uso de um sistema de armazenamento de dados em memória tem relação com o fato de que a gravação e leitura de dados de discos rígidos (dispositivo principal de armazenamento dos bancos de dados tradicionais) é milhares de vezes mais lenta do que a mesma operação utilizando a memória do computador (LAKE e CROWTHER, 2013, p. 183). Os bancos de dados em memória, neste caso, vêm facilitar a redução do acesso ao disco a fim de melhorar o desempenho de sistemas que os utilizem.

Figura 2 - Fluxo DBMS Tradicional x IMDB



Mas mesmo com o armazenamento em memória possuindo suas vantagens, principalmente em velocidade, eles não podem ser considerados a solução definitiva, pois, ainda precisam ser complementados por um sistema de armazenamento em disco para gravar os dados de modo permanente.

Quanto à sua classificação, os bancos de dados NoSQL podem ser categorizados em quatro classes conforme suas otimizações: chave-valor (*key-value*), documento (*document*), família de colunas (*column family* ou *columnar*) e banco triplo (*graph database* ou *triple*).

É possível classificar como do tipo chave-valor aqueles bancos de dados cuja a informação armazenada possui a sua respectiva chave. A Amazon utiliza seu sistema *key-value* Dynamo (DECANDIA, HASTORUN, *et al.*, 2007) para gerenciar funcionalidades tais como: listas de mais vendidos, carrinhos de compras, preferências do consumidor, gerenciamento de produtos, entre outras aplicações. A maioria dos bancos de dados que possuem armazenamento em memória são do modelo chave-valor principalmente pela simplicidade.

Já os bancos orientados a documentos (*documents*) resultaram, por exemplo, no MongoDB (MONGODB, 2009) e CouchDB (COUCHDB, 2005), onde as informações são armazenadas em uma estrutura de árvore, em formatos como XML ou JSON. O governo de Chicago utiliza uma plataforma para gerenciar operações

inteligentes construída sobre MongoDB denominada WindyGrid (CRAWFORD e GOLDSMITH, 2014). Esta ferramenta foi projetada para trabalhar com o grande volume dados gerados pela cidade. Através de um mapa de Chicago, é possível visualizar informações como chamadas para os serviços 911 e 311, trânsito, informações de construções, *tweets* públicos e outros dados críticos (THORNTON, 2013).

Os bancos de dados de família de colunas (*column family* ou *columnar*), são similares à estrutura tradicional de tabelas dos bancos de dados relacionais, porém, otimizados organizando as informações armazenadas em colunas

Os bancos classificados nesse modelo foram influenciados pelo *BigTable* da Google (CHANG, DEAN, *et al.*, 2008), tais como Cassandra (APACHE SOFTWARE FUNDATION, 2008) e HBase (APACHE SOFTWARE FOUNDATION, 2008). O Cassandra é utilizado pelo eBay para diversas funcionalidades, tais como: armazenar as notificações de usuários, gerir os itens das páginas, indicação de favoritos entre outras aplicações (LUBOW e BRADBERRY, 2013).

Por fim, os bancos de dados de grafos ou triplos (*graph database* ou *triple*) armazenam as informações compostas por três elementos: sujeito, propriedade ou relacionamento e valor (FOWLER, 2015) (KABAKUS e KARA, 2016). Deste modelo é possível citar como exemplo o Neo4j (NEO4J, 2007) e o JanusGraph (JANUSGRAPH, 2017).

Cada uma destas categorias traz sistemas que atendem à diferentes limitações dos bancos relacionais tradicionais e se complementam entre si. Nas próximas seções serão apresentadas as descrições e características de algumas tecnologias, de cada um dos tipos de bancos de dados NoSQL citados, os quais foram determinados a partir dos trabalhos relacionados, mesclando tecnologias já estudadas por outros autores, agregando assim possibilidades comparativas.

2.5.1 Bancos de dados chave-valor

Os bancos de dados NoSQL da variedade chave-valor são os representantes com as estruturas mais simples dentre os existentes (POKORNY, 2013) e possibilita a visualização da base de dados como uma tabela *hash*. No entanto, cabe ressaltar que eles possuem um amplo espectro de casos de uso, sendo largamente adotados,

tanto como armazenamento primário, quanto para auxiliar outros sistemas de armazenamento (CARLSON, 2013).

Neste modelo o armazenamento dos dados é a combinação de um conjunto de chaves, e estas estão ligadas a um valor, *string* ou binário. No Figura 3, é possível visualizar um exemplo da organização dos dados pessoais em um banco chave-valor. Na primeira coluna está o nome da chave e na segunda o valor que a mesma guarda.

Figura 3 - Exemplo de comandos de um banco de dados chave-valor

```
> SET “nome” “João da Silva”
OK
> GET “nome”
“João da Silva”
```

Este modelo, pode ser considerado de fácil integração, possibilitando assim que os dados sejam rapidamente acessados através da sua chave, contribuindo além disso para aumentar a disponibilidade de acesso as informações (FOWLER e SADALAGE, 2013, p. 213). A manipulação dos dados de modo geral é muito simples, geralmente sendo baseada em comandos como o *get*() e o *set*( ), que tem por função obter e entrar valores. Uma desvantagem deste modelo é que o mesmo não permite a recuperação de objetos através de consultas mais complexas, como pesquisas com *join*, por exemplo.

2.5.1.1 Redis

Redis (*REmote DIctionary Server*) (REDIS, 2009) é um sistema de armazenamento de dados estruturados em memória que pode ser utilizado como banco de dados, *cache* e *message broker* (CAO, SAHIN, *et al.*, 2016). Ele opera em um modelo cliente-servidor através de conexões TCP utilizando um protocolo próprio chamado RESP (*REdis Serialization Protocol*).

O modelo de dados do Redis é composto por cinco estruturas de dados diferentes para os valores (*string*, *list*, *set*, *sorted set* e *hash*), a persistência dos dados da memória em disco através de dois métodos (*snapshopts* chamados RDB e *append-only file* ou AOF, análogo ao método de *journaling* dos bancos tradicionais) (ZHANG, TAN, *et al.*, 2015). A possibilidade de escalabilidade horizontal através do Redis Cluster foi adicionada apenas em 2015, na versão 3.0 do sistema. Segundo os autores de (SANFILIPPO, 2010), um sistema deve ser eficiente em um único nó quando for escalado.

2.5.1.2 Memcached

O Memcached (MEMCACHED) caracteriza-se como um sistema genérico de *cache* em memória. Ele foi construído pensando na melhoria de desempenho de aplicações *web* através da redução na demanda de requisições ao banco de dados em disco. Brad Fitzpatrick desenvolveu ele para melhorar o desempenho do site Livejournal.com através de uma solução melhor de *cache* (GALBRAITH, 2009). A sua implementação é na linguagem Perl e posteriormente reescrito em C. O Memcached utiliza uma arquitetura *multi-thread* e o controle de concorrência interno é feito através de uma *hash-table* estática de *locks* (ZHANG, TAN, *et al.*, 2015).

A classificação do Memcached como banco de dados é discutível, uma vez que o mesmo não implementa persistência, *failover* (GALBRAITH, 2009) nem escalabilidade horizontal, pois a distribuição dos dados entre múltiplas instâncias do sistema deve ser feita pelo cliente (ZHANG, TAN, *et al.*, 2015). O funcionamento do Memcached segue o modelo cliente-servidor, e a comunicação ocorre através de conexões TCP ou UDP utilizando um protocolo próprio que suporta textos puros em ASCII ou dados binários (SOLIMAN, 2013).

2.5.1.3 Voldemort

O Voldemort (PROJECT VOLDEMORT) foi desenvolvido pelo LinkedIn em linguagem Java com o objetivo de gerenciar funcionalidades dependentes de associações entre dados da rede social, tais como a recomendação de relacionamentos através da análise dos relacionamentos atuais (SUMBALY, KREPS, *et al.*, 2012).

O Voldemort é inspirado no Dynamo, da Amazon (DECANDIA, HASTORUN, *et al.*, 2007), oferece comandos simples (*put*, *get* e *delete*) (SUMBALY, KREPS, *et al.*, 2012) e uma arquitetura completamente distribuída, onde cada nó é independente e não existe um servidor principal de coordenação (DEKA, 2014). O sistema é completamente modularizado, e tanto a serialização dos dados quanto a persistência são oferecidas através de módulos plugáveis. Segundo (SUMBALY, KREPS, *et al.*, 2012), a grande vantagem do Voldemort em relação ao Dynamo, é um mecanismo próprio de armazenamento desenhado para o pré-carregamento de grandes volumes de dados, em que o Voldemort passa a funcionar em modo somente leitura.

2.5.1.4 Aerospike

O Aerospike (AEROSPIKE, 2012) tem uma arquitetura modelada com foco em velocidade na análise de dados, escalabilidade e confiabilidade para aplicações web. Esse banco de dados se apresenta como uma solução para a combinação de diferentes tipos de dados e também acessos por milhares de usuários. Pensando nisso, as suas operações são focadas em chave-valor e otimizadas para o uso da memória RAM em conjunto com memórias flash (NVM).

Um dos pontos de destaque do Aerospike é a grande quantidade de bibliotecas de integração desenvolvidas pela própria empresa, a fim de melhorar o desempenho na sua utilização. Quanto ao *cluster*, todos os nós são iguais, em uma arquitetura conhecida como *shared nothing*. A respeito do *server* é possível utilizar índices secundários e definir funções para otimizar a utilização dos dados. E por fim, a camada de armazenamento incorpora a utilização da memória RAM e de sistemas de armazenamento permanente.

2.5.1.5 Hazelcast

O Hazelcast (HAZELCAST, 2009) é uma ferramenta distribuída sob licença *open source* e comercial desenvolvida em Java. Possui seu foco em computação distribuída e escalabilidade horizontal, se destacando dos concorrentes por oferecer as garantias ACID (Atomicidade, Consistência, Isolamento e Durabilidade) dos bancos de dados relacionais tradicionais.

O *cluster* funciona em uma arquitetura *shared nothing*, onde não existe um ponto único de falha. Além de oferecer clientes para as linguagens comuns como Java, C, C++ e C#, o Hazelcast oferece uma API REST, está preparado para trabalhar com o protocolo de comunicação do Memcache e pode ser utilizado através do Hibernate (HAZELCAST, 2009).

2.5.1.6 Riak KV

O Riak KV (BASHO, 2009) possui como principal objetivo oferecer disponibilidade máxima, com escalabilidade horizontal em forma de cluster, sendo considerado um banco de dados de simples operação e fácil escalabilidade. Em sua versão comercial há suporte a *multi-cluster replication*, ou seja, é possível realizar a

replicação de dados através de diferentes clusters, geograficamente distantes, a fim de reduzir a latência de acesso uniformemente para clientes através do globo.

Nota-se claramente a influência do Dynamo (DECANDIA, HASTORUN, *et al.*, 2007) no Riak KV, desde suas funcionalidades para execução distribuída até nas configurações do fator de replicação e arquitetura *shared nothing*.

2.5.2 Bancos de dados de famílias de colunas

Os bancos de dados representantes do modelo família de colunas ou *columnar* são aqueles que mantém as informações armazenadas em colunas do banco de dados separadamente, guardando junto com os valores de atributos pertencendo à mesma coluna, de modo denso e comprimido.

Embora o modelo de colunas compartilhe o conceito de armazenamento dos sistemas relacionais, a diferença é o modo em que os dados não ficam armazenados em tabelas, mas sim em arquiteturas que estão distribuídas massivamente. No armazenamento em colunas, cada chave está associada a um ou mais atributos (colunas), a Figura 4 ilustra esta teoria. Neste tipo de banco de dados as informações ficam guardadas de tal modo que as informações possam ser agregadas rapidamente com menor atividade de E/S (NAYAK, PORIYA e POOJARY, 2013).

Figura 4 - Organização da base de dados *columnar*

Fonte: (POLLOCK, 2011, p. 99)

Os bancos colunares encorajam o uso da desnormalização, isso é possível realizando a cópia para vários registros. Com isso é possível obter um ganho na velocidade de leitura, pois não são utilizados relacionamentos que demandam um trabalho de reconstituição dos dados (FOWLER, 2015). Porém uma desvantagem é que com isso a perda de velocidade na inserção (NAYAK, PORIYA e POOJARY, 2013).

2.5.2.1 BigTable

O BigTable (GOOGLE, 2005) é um sistema de armazenamento distribuído, desenvolvido para gerenciar dados estruturados e com a possibilidade de escalabilidade muito grande de dados entre milhares de servidores de *commodities*. Diversos projetos do Google armazenam dados no BigTable, incluindo a indexação da Web, Google Earth e Google Finance (CHANG, DEAN, *et al.*, 2008).

O BigTable foi projetado para trabalhar em um cenário com cargas de trabalho maciças em baixa latência consistente e alta taxa de transferência, por isso é uma execelente opção para *softwares* operacionais e analíticos, incluindo para IoT, análise de usuários e análise de dados financeiros. (GOOGLE).

Na arquitetura do BigTable existe uma camada denominada *memtable,* que fica responsável por armazenar as entradas de log, que são escritas e mantidas na memória principal. A arquitetura é complementada com as SSTables que atuam como um instantâneo do estado de um *tablet,* e caso ocorra uma falha, a recuperação é realizada pela reprodução das entradas de *log* mais recentes desde o último instantâneo. Já as leituras são executadas através uma visão combinada dos dados das SSTables com a *memtable* (CHANG, DEAN, *et al.*, 2008).

2.5.2.2 HBase

O HBase (APACHE SOFTWARE FOUNDATION, 2008) é um banco de dados que possui como principais características o fato de ser *open-source*, distribuído, versionado, se enquadra na estrutura NoSQL e foi desenvolvido com base o BigTable (GOOGLE, 2005), porém otimizado sob o Hadoop (APACHE, 2011) e o Hadoop *Distributed File System* (HDFS) (APACHE, 2011).

O seu fornecedor recomenda este banco de dados para aqueles que necessitam de acesso aleatório as informações, em tempo real e com demanda de velocidade na leitura e gravação de seus dados. O foco deste projeto é manter tabelas com grandes volumes de dados, bilhões de linhas e milhões de colunas, isso sob *clusters* de *hardware* de *commodities* (APACHE SOFTWARE FOUNDATION, 2008).

Como o HBase estende o modelo BigTable (GOOGLE, 2005), ele considera um único índice, isto é semelhante a uma chave primária nos modelos de banco de dados relacionais. Deste é possível fornecer ganchos do lado do servidor para implementar soluções flexíveis de índice secundário. Além disso, ele oferece predicados *push-down*, que se tratam de filtros de que permitem reduzir os dados trafegados na rede (GEORGE, 2011).

2.5.2.3 Cassandra

O Apache Cassandra (APACHE SOFTWARE FUNDATION, 2008) é um projeto de sistema de banco de dados distribuído que se enquadra no modelo NoSQL, além disso é classificado como escalável, e reúne a arquitetura do DynamoDB (DECANDIA, HASTORUN, *et al.*, 2007) e modelo de dados baseado no BigTable (GOOGLE).

O Cassandra foi desenvolvido pelo Facebook, o que abriu seu código-fonte para a comunidade em 2008. Agora é mantido por desenvolvedores da fundação Apache e colaboradores de muitas empresas. (HAN, HAIHONG, *et al.*, 2011)

Este banco de dados foi desenvolvido em linguagem Java, e suporta multi plataformas. Na questão da durabilidade dos dados ele trabalha com *CommitLog*, onde ficam salvas todas as mutações locais a um nó de Cassandra. Quaisquer dados escritos para Cassandra serão primeiro escritos em um log de confirmação antes de serem gravados em um *memtable*. (APACHE SOFTWARE FUNDATION, 2008)

2.5.2.4 Accumulo

Apache Accumulo (APACHE ACCUMULO, 2008) é baseado no modelo proposto pelo Google BigTable (GOOGLE) e é carregado pelo Apache Hadoop, Apache Zookeeper e Apache Thrift. O Accumulo destaca em sua página recursos como a função de controle de acesso baseado em células e um mecanismo de programação do lado do servidor, que promete possibilitar a modificação dos pares

de chave / valor em vários pontos no processo de gerenciamento de dados (APACHE ACCUMULO, 2008).

O Accumulo é escrito em Java e roda sob o *Hadoop Distributed File System* (HDFS) (APACHE, 2011). Este banco de dados suporta o armazenamento e a recuperação de dados estruturados, incluindo consultas para intervalos, e fornece suporte para o uso de tabelas Accumulo como entrada e saída para MapReduce (DEAN e GHEMAWAT, 2008).

2.5.3 Bancos de dados orientados a documentos

Os bancos de documentos são semelhantes aos bancos de chave-valor, porém neste caso o valor é estruturado e possui hierarquia. Diferentemente dos bancos tradicionais, estes não possuem um esquema ou estrutura pré-definida (FOWLER, 2015).

É bastante comum a utilização dos formatos JSON e XML para representar os documentos nestes tipos de bancos de dados (POKORNY, 2013) e o suporte ao processamento e análise de dados é bastante variado dentre os representantes da categoria. Uma característica interessante é o método *append-only*, que é utilizado em alguns bancos de documentos para armazenar os dados e oferece operações de escrita em tempo constante.

2.5.3.1 MongoDB

O MongoDB (MONGODB, 2009) é o representante mais conhecido dentre todos os bancos NoSQL, não apenas de sua categoria (YUHANNA, LEGANZA e AUSTIN, 2016). Os documentos são armazenados em um formato binário conhecido como BSON (*Binary JSON*), mas os clientes gerenciam a conversão para o protocolo JSON de forma transparente (FOWLER, 2015).

Seguindo à risca o paradigma NoSQL, o MongoDB não oferece nenhum suporte nativo a relacionamentos entre documentos ou consultas com os populares *JOIN* dos bancos relacionais, defendendo a de-normalização dos dados (onde os dados são armazenados na forma em que são necessários na consulta) de forma a maximizar o desempenho das consultas (COPELAND, 2013).

2.5.3.2 CouchDB

O CouchDB (COUCHDB, 2005) possui foco na facilidade de uso, oferecendo a flexibilidade dos bancos de documentos e comunicação através de uma API HTTP RESTful. Algumas das características mais marcantes do CouchDB são o suporte completo à transações ACID, o foco em disponibilidade e a possibilidade de sincronização de réplicas para uso off-line, porém nota-se o pouco foco na utilização da memória RAM para melhorias no desempenho do sistema.

As consultas podem ser feitas em documentos individuais ou através da construção de “visões” definidas por funções JavaScript e indexadas pelo CouchDB de forma a melhorar o desempenho de consultas.

2.5.3.3 Couchbase

O Couchbase (COUCHBASE, 2010), originalmente chamado de Membase, é o produto da empresa homônima, formada através de uma fusão entre membros dos projetos memcached e CouchDB.

Ele oferece a velocidade e as garantias de consistência normalmente encontradas em bancos chave-valor, sendo considerado um dos bancos de documentos mais rápidos (FOWLER, 2015).

2.5.3.4 MarkLogic

O MarkLogic (MARKLOGIC, 2001) é o mais antigo dentre os bancos NoSQL avaliados, com mais de 15 anos de existência, podendo ser classificado tanto como banco de documentos quanto como banco triplo, além de um motor de busca. É considerado um dos líderes dentre os bancos de documentos, assim como MongoDB e Couchbase (YUHANNA, LEGANZA e AUSTIN, 2016).

A maturidade do MarkLogic é facilmente percebida pela extensa lista de funcionalidades empresariais oferecidas, além de ser o único banco NoSQL com certificação de segurança de software expedida pelo departamento de defesa dos Estados Unidos (FOWLER, 2015).

2.5.4 Bancos de dados de grafos ou triplos

Os bancos triplos possuem foco na criação de relacionamentos entre os dados. Porém, diferentemente dos bancos relacionais tradicionais, os dados não são tabulares, e sim armazenados em registros triplos compostos por sujeito, predicado e objeto.

Outra nomenclatura adotada para estes bancos e seus dados é a sigla RDF ou *Resource Description Framework*, onde os dados são compostos por recurso, propriedade e indicação ou valor.

Com estes bancos é possível construir redes complexas de relacionamentos e revelar novos dados através da inferência. Bancos triplos com suporte a consultas complexas, tais como análise de caminhos e distâncias entre dois nós, são conhecidos como bancos de grafos (FOWLER, 2015).

O projeto *Apache TinkerPop* (APACHE TINKERPOP, 2008) tem buscado padronizar a camada de comunicação entre aplicações e bancos de grafos, através da oferta de um framework que inclui uma linguagem de navegação em grafos chamada *Gremlin*.

2.5.4.1 Neo4j

O Neo4j (NEO TECHNOLOGY, 2007) é o banco de grafos mais popular atualmente, possuindo código-fonte aberto e oferecendo suporte a transações ACID e ao framework *TinkerPop*. A memória principal é utilizada principalmente para cache, e o banco trabalha completamente em memória se o tamanho do cache for maior que o tamanho dos dados.

Os casos de uso mais comuns do Neo4j incluem sistemas de recomendações em tempo real, busca baseada em grafos, redes sociais, detecção de fraudes, gerenciamento de rede e identidade, dentre outros (YUHANNA, LEGANZA e AUSTIN, 2016).

2.5.4.2 OrientDB

O OrientDB (ORIENTDB, 2010) destaca-se por ser o primeiro banco NoSQL com abordagem multi-modelo, funcionando como banco triplo, banco de documentos em JSON e banco chave-valor, com suporte ao *TinkerPop*.

Ainda que o OrientDB não seja considerado um líder de mercado, é considerado um competidor de peso por seu baixo custo, facilidade de uso, bom desempenho, amplo espectro de casos de uso e baixa curva de aprendizado por oferecer funcionalidades como transações ACID e suporte à linguagem SQL (YUHANNA, LEGANZA e AUSTIN, 2016).

2.5.4.3 JanusGraph

O JanusGraph foi criado em 2017 com base no último código-fonte disponível do Titan (TITAN, 2012), que havia sido abandonado em 2015 após a venda da empresa Aurelius, responsável pelo sistema, para a DataStax. O JanusGraph caracteriza-se como um banco de grafos otimizado para escalabilidade horizontal, alta concorrência e grandes volumes de dados, com integração a várias outras tecnologias através de módulos plugáveis. Como toda a camada de armazenamento é oferecida através de módulos integrados, muitas de suas características de distribuição e replicação de dados para melhoras no desempenho e tolerância a falhas, alta disponibilidade e suporte às garantias ACID dependem da escolha da tecnologia responsável por esta camada.

No momento em que esta pesquisa foi realizada, o JanusGraph oferecia módulos para integração (*storage backends*) com os bancos Cassandra, HBase, BerkeleyDB e um módulo de processamento de dados em memória.

A comunicação e o modelo de dados do JanusGraph é baseado no *TinkerPop*, não possuindo outra linguagem ou forma de interação. Uma das características marcantes do JanusGraph é o suporte ao acoplamento de várias tecnologias, tanto para armazenamento quanto para processamento de dados, oferecendo funcionalidades como consultas baseadas em georreferenciamento e pesquisas completas em texto (*full-text search*).

2.5.4.4 Graph Engine

O Graph Engine (MICROSOFT, 2017), originalmente chamado de Trinity, possui foco na utilização da memória RAM combinada de máquinas conectadas em cluster. Sua camada de armazenamento possui a estrutura de um banco chave-valor, onde cada par chave-valor (aqui chamada de célula) representa um nó (*node*) ou uma

aresta (*edge*) do grafo (SHAO, WANG e LI, 2013). As arestas podem conter rótulos ou predicados, compondo assim um banco RDF.

Destaca-se por ser um dos únicos bancos de grafos que não possui suporte ao *TinkerPop*. O armazenamento dos dados é feito em um formato binário serializado, o que reduz o uso de memória e facilita a localização de dados, mas também dificulta o processo de análise de dados porque inclui a necessidade de serialização e deserialização de dados (ZHANG, TAN, *et al.*, 2015).

2.5.4.5 Bitsy

O Bitsy (RAMACHANDRAN, 2013) é um banco de grafos NoSQL sem suporte à distribuição ou escalabilidade horizontal. Ele deve ser utilizado embutido diretamente na aplicação Java, armazenando seus dados inteiramente na memória principal e gravando um log destes dados em disco para recuperação em caso de falha.

Os principais direcionadores do Bitsy são a eliminação de buscas e leituras do disco rígido, utilizando um arquivo de log *append-only* para persistência; a eliminação da comunicação via rede, funcionando como um banco embutido na própria aplicação; e a eliminação de consultas SQL, utilizando a API Blueprints para trabalhar com grafos (ZHANG, TAN, *et al.*, 2015).

O desenvolvimento do Bitsy ficou parado durante três anos (de 2013 a 2016), mas uma versão 2.0 com suporte a escalabilidade horizontal em cluster está planejada.

2.6 TRABALHOS RELACIONADOS

A construção do conhecimento científico e a evolução da ciência ocorre através de uma metodologia incremental, onde é feito o estudo do conhecimento já disponível e só então são propostas novas formas e linhas de pesquisa para a construção de novos conhecimentos. Como bem colocou Isaac Newton em 1676, “se eu vi mais longe, foi porque estava sobre os ombros de gigantes”.

Esta seção apresenta as pesquisas relacionadas que antecederam e basearam este estudo. No Quadro 1 – Trabalhos relacionados estão descritas as pesquisas,

seus autores, assim como suas contribuições, lacunas e diferenças se comparadas a esta. Na primeira coluna consta os autores e o ano em que o trabalho foi publicado, na segunda coluna as tecnologias abordadas, na sequência está apresentado a classificação que as tecnologias se enquadram dentro do âmbito NoSQL, por fim algumas observações.

Quadro 1 – Trabalhos relacionados

Estudo	**Tecnologias**	**Foco**	**Observações**
Han, et al., 2011	Redis, Tokyo, Flare	NoSQL key-value, columnar e document	Não oferece comparativo.
Hecht and Jablonski, 2011	Voldemort, Redis, Membase	NoSQL	Avalia API, concorrência, replicação e consistência.
Deka, 2014	Hypertable, Voldemort, Dynomite, Redis, Dynamo	NoSQL	Características mercadológicas.
Zhang, et al., 2015	MemepiC, RAMCloud, Redis, Memcached, MemC3, TxCache	IMDB	Avalia o projeto, modelo, índices, concorrência, etc
Cao, Sahin, et al., 2016	Redis, Memcached	IMDB key-value	Se off a persistência em disco, Redis superior ao Memcached.
Kabakus e Kara, 2016	MongoDB, Redis, Memcached, Cassandra, H2	NoSQL	Não detalha o modelo de persistência; Memcached superior ao Redis.
Abramova, Bernardino e Furtado, 2014	Cassandra, Hbase, MongoDB, OrientDB, Redis	NoSQL	Desempenho de leitura, modificação e inserção. Não considera o ambiente.
Anderle, Rockenbach, *et al.*, 2017	Redis, Memcached, Voldemort, Aerospike, Hazelcast, Riak KV, BigTable, HBase, Hypertable, Accumulo, MongoDB, CouchDB, Couchbase, MarkLogic, OrientDB, Neo4j, JanusGraph, Graph Engine, Bitsy	NoSQL	Características mercadológicas, projeto, manutenção e avaliação de desempenho (tecnologias e ambiente)

2.6.1 Trabalhos relacionados ao estudo do estado da arte

Nesta seção é apresentada uma discussão sobre os trabalhos relacionados publicados recentemente na literatura que abordam estudos do estado da arte. De forma semelhante, todos os trabalhos selecionados procuram caracterizar e comparar bancos de dados NoSQL, sendo complementados por outras classificações de bancos de dados ou não.

Tanto (HECHT e JABLONSKI, 2011) quanto (HAN, HAIHONG, *et al.*, 2011) se propõem a fazer um estudo e avaliação dos bancos de dados NoSQL, com o mesmo objetivo principal: prover informações para auxiliar na escolha do banco NoSQL que melhor atende às necessidades. No ponto de intersecção entre este trabalho e os citados, (HECHT e JABLONSKI, 2011) inclui em seu trabalho os bancos Project Voldemort, Redis e Membase, enquanto que (HAN, HAIHONG, *et al.*, 2011) avalia Redis, Tokyo Cabinet-Tokyo Tyrant e Flare.

Em (DEKA, 2014) é apresentada uma visão geral de quinze sistemas NoSQL, tais como, Cassandra, BigTable, Pnuts, Redis, entre outros. Nota-se a falta, porém, de uma visão comparativa mais clara sobre aspectos de garantias de durabilidade, disponibilidade, protocolos suportados, e outras informações que podem vir a influenciar significativamente na escolha de um banco de dados.

Já (ZHANG, TAN, *et al.*, 2015) traz uma visão bem estruturada dos objetivos que nortearam o projeto de cada um dos sistemas descritos na pesquisa, o qual foca em sistemas com gerenciamento e processamento de dados em memória. Dentre os sistemas estudados, os representantes NoSQL são MemepiC, RAMCloud, Redis, Memcached, MemC3 e TxCache. O autor descreve no artigo as cargas de dados mais adequadas aos sistemas, a estratégia para construção de índices, o controle de concorrência, tolerância a falhas, tratamentos para conjuntos de dados maiores do que a memória disponível e o suporte a consultas personalizadas em baixo nível (como *stored procedures* e *scripts* em linguagem nativa), porém com pouca abordagem de alto nível que auxilie na escolha de um sistema em favor de outro.

2.6.2 Trabalhos relacionados a avaliação de desempenho

Nesta seção é apresentada uma discussão sobre os trabalhos relacionados publicados recentemente na literatura que abordam estudos referentes a avaliação de desempenho de bancos de dados. De forma semelhante, todos os trabalhos selecionados procuram caracterizar e comparar bancos de dados NoSQL, sendo complementados por outras classificações de bancos de dados ou não.

O artigo “*Evaluation and Analysis of In-Memory Key-Value Systems*” traz uma avaliação comparativa entre os bancos de dados em memória Redis e Memcached, com *benchmarks* aplicados em uma única plataforma servidora com CPU Intel Core i5-4460 de 3.20GHz, 16GB de memória RAM DDR3 de 1600MHz e um disco SSD

Samsung SATA-3 de 250GB, rodando um sistema operacional Linux Ubuntu 14.04 (CAO, SAHIN, *et al.*, 2016).

O trabalho "*A performance evaluation of in-memory databases*", por sua vez, envolveu a comparação das soluções Redis, MongoDB, Memcached, Cassandra e H2 em uma única máquina com CPU Intel Core i7-4710MQ de 2.5GHz, 16GB de memória RAM rodando um sistema operacional Linux Ubuntu 14.04 (KABAKUS e KARA, 2016).

O artigo "*Which NoSQL Database - a performance overview*", realizou a comparação de desempenho de cinco bancos de dados, considerados pelos autores como populares dentre a comunidade NoSQL, são eles: Cassadra, HBase, MongoDB, OrientDB e Redis. Os autores realizaram o estudo utilizando o *benchmark* YCSB (YAHOO, 2010), e os resultados obtidos, foram apenas em relação a velocidade de leitura, modificação e inserção de cada sistema com cada uma das cargas de trabalho definidas. (ABRAMOVA, BERNARDINO e FURTADO, 2014)

Além dos trabalhos científicos, é válido citar os *benchmarks* aplicados pelo idealizador do Redis, Salvatore Sanfilippo (2010) e pelo principal colaborador do memcached, Dormando (2010), cada qual apontando a sua solução como tendo o melhor desempenho.

A falta de dados claros sobre o ambiente e as configurações utilizadas para os testes também são um problema, como por exemplo o trabalho de Kabakus e Kara (2016), que falha ao não detalhar o modelo de persistência utilizado nos testes com o Redis, apontando o Memcached como superior em quase todos os aspectos. A importância desta informação é evidente ao analisar os resultados obtidos por Cao, Sahin, *et al.* (2016), onde o Redis possui desempenho superior ao Memcached em algumas situações, quando a persistência é desativada.

Ainda que os *benchmarks* aplicados por Sanfilippo (2010) e Dormando (2010) destaquem, respectivamente, o desempenho do Redis e do Memcached, todas estas pesquisas podem ser consideradas tendenciosas, visto que os autores fizeram parte do time de desenvolvimento de alguma das ferramentas envolvidas no estudo.

Além disso, nenhum dos trabalhos relacionados abrange uma gama expressiva de ferramentas quanto o presente estudo, nem apresenta uma análise comparativa

das funcionalidades oferecidas por cada uma delas, limitando seu uso efetivo como ferramenta de auxílio à tomada de decisão.

CAPÍTULO 3: RESULTADOS OBTIDOS

3.1 ALTERAÇÃO DO ESCOPO

O projeto da presente pesquisa foi elaborado e entregue sem uma profunda revisão da literatura, pois o cronograma do trabalho de conclusão não abrange tal necessidade. Diante disso, por se tratar de um trabalho científico (e não uma prática profissional ou estágio), durante uma análise mais profunda no andamento da pesquisa percebeu-se que o cronograma proposto não permitiria a completude da pesquisa e, portanto, verificou-se a necessidade de um alinhamento da proposta inicial e um direcionamento voltado mais para o cenário de banco de dados.

De acordo com o descrito na proposta de trabalho e apresentado no Apêndice F, o escopo inicial previa que os testes de desempenho das ferramentas de bancos de dados em memória e *message brokers* fossem realizados de forma independente e comparando os resultados apenas entre as categorias, ou seja, bancos de dados comparados com bancos de dados, e *message brokers* com *message brokers*.

Porém, com o estudo do estado da arte em relação aos bancos de dados completo, notou-se que os resultados finais da pesquisa não se complementariam, e a pesquisa iria possuir dois focos desconexos. Também identificou-se uma gama expressiva de bancos de dados classificados como NoSQL, e dentre estes bancos foi possível notar que os bancos de dados em memória são na sua grande maioria aqueles classificados como chave-valor.

Diante da impossibilidade de seguir todas as linhas de pesquisa, optou-se por direcionar o foco dos testes de desempenho aos bancos de dados NoSQL, incluindo na pesquisa também bancos de dados *memory-aware* que não necessariamente possuem a memória RAM como armazenamento principal. Com isso foi possível dar

profundidade ao estudo em relação a banco de dados, conforme espera-se de uma pesquisa.

3.2 LARCC

O LARCC (*Laboratory of Advanced Researches for Cloud Computing)*, está atualmente localizado no Laboratório de Redes da Sociedade Educacional Três de Maio (SETREM), e possui como coordenador o Prof. Dr. Dalvan Griebler. O principal objetivo é disponibilizar serviços de infraestrutura para a viabilizar a execução de pesquisas científicas na Faculdade, e permitir que os acadêmicos tenham um ambiente para aprendizado prático no cenário de computação em nuvem (SETREM, 2015).

O laboratório permite aos acadêmicos que apliquem pesquisas, as quais demandam de uma infraestrutura de computadores que tenham alto desempenho para simular situações, testar *softwares*, analisar dados, e as mais diversas aplicações. Além disso, o laboratório busca parcerias para executar projetos de pesquisa em cooperação com organizações da região com a finalidade de auxiliar na resolução de problemas e propor alternativas para outras situações.

As possibilidades de pesquisa que o LARCC pode contribuir são: infraestrutura como serviço (IaaS), computação em nuvem, plataforma como serviço (PaaS), sistemas distribuídos, software como serviço (SaaS), eficiência energética para *datacenter* e nuvem privada, programação para redes (protocolos de aplicação), mineração de dados e aprendizado de máquina para agricultura.

O LARCC oferece atualmente um serviço de nuvem IaaS, sendo que esse possui o objetivo de disponibilizar um ambiente com infraestrutura de *hardware* para implementação de virtualização. O ambiente de produção, que é utilizado pelas organizações parceiras no LARCC, não é o mesmo que a comunidade acadêmica utiliza para execução de experimentos.

O ambiente de experimentos é composto por oito máquinas, sendo que todas rodam o sistema operacional Ubuntu Server 14.04, o qual foi aplicado por se tratar de uma distribuição com uma ampla documentação principalmente em relação ao gerenciamento de ambientes em nuvem.

3.3 ANÁLISE COMPARATIVA DOS BANCOS DE DADOS NOSQL

Esta seção apresenta uma comparação dentre as tecnologias avaliadas em cada categoria de banco de dados, com suas respectivas características. Para facilitar a análise, as características foram classificadas em três grupos: características mercadológicas, características do projeto e características de manutenção.

Nas características mercadológicas são explorados aspectos sem relação direta com funcionalidades ou o funcionamento do banco, tais como o ano em que a primeira versão do sistema foi lançada, os licenciamentos sob os quais o software é distribuído, a linguagem na qual o sistema foi desenvolvido, os sistemas operacionais suportados, as linguagens nas quais são oferecidos clientes para comunicação e os protocolos de comunicação suportados. A partir destes dados os interessados podem avaliar a maturidade do sistema, se a licença está alinhada com as necessidades empresariais, e se a infraestrutura disponível e o esforço de implementação estão dentro do esperado.

Nas características do projeto são descritas definições decididas no projeto do sistema, tais como as opções para escalabilidade horizontal (ou clusterização), a classificação do sistema segundo o teorema CAP (explicado no item 2.1.12), como é feito o controle de concorrência, o suporte à transações ACID, as opções de persistência dos dados em disco, o suporte a dados complexos e as opções para autenticação do cliente. Analisando-se estas características é possível avaliar se as funcionalidades e características do banco de dados atendem às demandas e características referentes aos dados que se pretende armazenar nos mesmos

Nas características de manutenção são explanados aspectos que tem relação direta com a manutenção e suporte ao sistema, tais como a existência de interfaces de gerenciamento, ferramentas de monitoramento e *benchmarks* embutidos (que são disponibilizados junto com o sistema e cujo principal objetivo é avaliar o desempenho do sistema em determinada infraestrutura) para os sistemas estudados. É importante notar que foram avaliadas apenas as ferramentas oficiais dos desenvolvedores dos sistemas, portanto muitas destas ferramentas, ainda que não estejam declaradas aqui, já foram desenvolvidas pela comunidade e estão disponíveis. Estas informações são particularmente interessantes para avaliar o esforço de manutenção que será despendido após a implantação do sistema.

3.3.1 Bancos de dados chave-valor

Esta seção apresenta uma comparação entre as características mercadológicas (Quadro 2), de projeto (Quadro 3) e de manutenção (Quadro 4) das tecnologias chave-valor apresentadas na seção 2.5.1.

Ainda que o memcached possa ser considerado um dos precursores dos bancos de dados chave-valor modernos, o Redis é considerado o sistema mais popular da categoria (DB-ENGINES). A publicação do artigo que descreve o funcionamento do sistema Dynamo, da Amazon (DECANDIA, HASTORUN, *et al.*, 2007), pode ser apontado como um dos influenciadores para o surgimento de vários sistemas desta categoria a partir do ano de 2009.

Quadro 2 - Características mercadológicas dos bancos chave-valor

	Redis	**Memcached**	**Voldemort**	**Aerospike**	**Hazelcast**	**Riak KV**
Lançamento	2009	2003	2009	2012	2009	2009
Licença	BSD-3 e comercial	BSD-3	Apache 2	AGPL e comercial	Apache 2 e comercial	Apache 2 e comercial
Desenvolvido	C	C	Java	C	Java	Erlang
SO Suportados	Linux, BSD, OS X e Windows	Debian / Ubuntu e Windows	JVM	Linux, OS X e Windows	JVM	Linux
Qtde de Linguagens Suportadas	1 oficial e 47 da comunidade	Não existe listagem oficial	4 oficiais	12 oficiais	6 oficiais	8 oficiais e 20 da comunidade
Protocolos	Próprio (RESP)	Próprio	HTTP, Socket, NIO	Próprio e JDBC	Próprio e Memcached	API, HTTP e Próprio

A respeito dos sistemas operacionais, vale ressaltar que o Redis não suporta oficialmente Windows, mas uma versão para Windows x64 é mantida pela equipe da MS Open Tech (Microsoft Open Technologies). Quanto ao Voldemort e ao Hazelcast, como os mesmos rodam na JVM (*Java Virtual Machine*), podem-se considerar os sistemas operacionais suportados por esta tecnologia. Tanto Aerospike quanto Riak KV oferecem pacotes para sistemas baseados nas distribuições Linux Red Hat, Debian e Ubuntu. O Aerospike ainda oferece sua execução no OS X e Windows através de máquinas virtuais.

Quanto ao item linguagens com cliente, é importante notar que foram consideradas apenas as linguagens e clientes listados no site ou repositório oficial de cada sistema, e que o site do Memcached não oferece uma listagem oficial das

linguagens suportadas. Nota-se também que as linguagens C++, Java e Python são as únicas para as quais todos os sistemas possuem clientes listados.

Os protocolos de comunicação são o meio de comunicação entre o cliente e o servidor, porém, na maioria dos casos a comunicação é feita através de um dos clientes já construídos, e o usuário não precisa se preocupar com o protocolo utilizado pelo cliente.

Quadro 3 - Características do projeto dos bancos chave-valor

	Redis	**Memcached**	**Voldemort**	**Aerospike**	**Hazelcast**	**Riak KV**
Escabilidade horizontal	Mestre - Escravo	Não	Fator de Replicação	Fator de Replicação	Fator de Replicação	Fator de Replicação
Teorema CAP	CP	N/A	AP	AP	AP	Configurável
Controle de concorrência	Single thread	Mutex lock	MVCC	*Compare-and-swap*	Multi / single thread	MVCC
Transações	Parcial	Não	Não	Parcial	Sim	Não
Persistência em disco	RDB e AOF	Não	Config.	Assínc.	Banco Auxiliar	Banco auxiliar
Suporte a dados complexos	Sim	Não	Sim	Sim	Sim	Sim
Autenticação	Simples	SASL	Kerberos	Somente comercial	Simples, SSL, Kerberos, IP	Sim e autorização

Quanto à escalabilidade, todos os bancos exceto o Memcached incluem suporte à *sharding* e replicação, sendo que a maioria (a exemplo do Dynamo (DECANDIA, HASTORUN, *et al.*, 2007)) oferecem fator de replicação configurável para leituras e escritas. O Redis utiliza o modelo *master*/*slave*, comumente utilizado nas bases de dados relacionais tradicionais.

Quanto à classificação do Redis no teorema CAP, é importante mencionar que o ele não atende todos os requisitos de um sistema CP de (BREWER, 2000), por usar replicação assíncrona entre os nós do cluster. O teorema CAP também não se aplica ao Memcached pelo fato de ele não suportar a criação de clusters e, portanto, não haver comunicação entre nós. O Riak KV possui uma configuração onde é possível definir qual dos atributos devem ser preservados (disponibilidade ou consistência).

A classificação de Voldemort, Aerospike e Hazelcast como sistemas AP é válida apenas se for utilizado um fator de replicação igual a 1, para atender à definição

de disponibilidade do teorema CAP explicado no item 2.1.12. O Riak KV é o único que oferece linearizabilidade quando configurado para priorizar consistência, podendo, portanto, ser classificado como CP.

O controle de concorrência é implementado de diferentes maneiras. No Redis a execução é *single-thread* e as requisições são processadas de forma assíncrona internamente (ZHANG, TAN, *et al.*, 2015), sendo que apenas um *master* responde por uma determinada chave, portanto não há concorrência. O Memcached utiliza *mutex* (*mutual exclusive*) *lock*. Tanto Voldemort quanto Riak KV seguem a implementação do Dynamo (DECANDIA, HASTORUN, *et al.*, 2007) e utilizam *vector clocks*, uma implementação do versionamento baseado em bloqueio otimista (*optimistic locking*) conhecida por MVCC (*Multi Version Concurrency Control*). O Aerospike utiliza o método conhecido como CAS (*compare-and-swap*, *check-and-set* ou *test-and-set*), uma operação atômica implementada a baixo nível que escreve em um local de memória e retorna o valor antigo. No Hazelcast, é criada uma *thread* para atender cada uma das partições internas de dados, portanto ainda que ele seja *multi-thread*, uma chave específica está num contexto *single-thread* e, portanto, não há concorrência.

Quanto as transações, há um nível bastante variado de suporte oferecido pelos sistemas. Ainda que o Redis tenha suporte básico a transações, estas não possuem opção de *rollback*, e a durabilidade da mesma depende da persistência em disco. Memcached, Voldemort e Riak KV declaram não suportarem transações, enquanto que o Aerospike suporta transações que envolvam uma única chave ou que sejam somente leitura, no caso de envolverem múltiplas chaves. O Hazelcast se destaca sendo o único a oferecer transações ACID completas.

A persistência em disco é oferecida por todos os bancos, exceto o Memcached. No Redis são oferecidas duas formas complementares: RDB (*snapshot*), onde todos os dados na memória são gravados em disco, e *journal* (aqui chamado de AOF ou *append-only file*), onde cada operação é gravada em um arquivo de log e o arquivo é reescrito quando chega em um tamanho pré-determinado. O Voldemort oferece opções para configurar a persistência como síncrona (*write through*, onde a operação é persistida antes do retorno ao cliente) ou assíncrona (*write behind*, onde o cliente recebe a confirmação e posteriormente a operação é persistida), enquanto que o Aerospike faz a persistência de forma assíncrona. Vale mencionar que o Aerospike

tem melhorias focadas no uso de SSD como dispositivo de armazenamento permanente. Tanto Hazelcast como Riak KV oferecem persistência através do acoplamento de um banco de dados auxiliar e a assincronicidade é configurável.

Referente ao suporte a dados complexos, vale notar que todos os sistemas, exceto o Memcached, suportam chaves do tipo lista e *hash table* (ainda que com nomes diferentes). Hazelcast e Voldemort se baseiam fortemente nas classes do Java, Redis e Riak KV ainda oferecem suporte ao tipo *HyperLogLogs*, enquanto Redis e Aerospike oferecem suporte a tipagem ou comandos relativos a georreferenciamento.

Por final, enquanto o Redis oferece autenticação simples através de credenciais pré-configuradas, o Memcached oferece autenticação através do protocolo SASL (*Simple Authentication and Security Layer*). Já o Voldemort é integrado ao protocolo Kerberos. O Aerospike oferece autenticação apenas em sua versão com licenciamento comercial. O Hazelcast oferece todos os protocolos supracitados e ainda SSL. Por fim, o Riak KV oferece um sistema próprio de usuários e grupos, com autenticação e autorização baseada em diversos mecanismos, incluindo senhas e certificados digitais.

Quadro 4 - Características de manutenção dos bancos chave-valor

	Redis	**Memcached**	**Voldemort**	**Aerospike**	**Hazelcast**	**Riak KV**
Interface de Gerenciamento	Não	Não	Básica	À parte	Comercial	Sim
Ferramentas de Monitoramento	‘INFO’	‘stats’	JMX	‘asadm’	JMX	‘stats’
Benchmark embutido	Sim	Não	Sim	Sim	Não	Sim

Quanto às interfaces de gerenciamento oferecidas pelos sistemas avaliados, o Redis e o Memcached são os únicos que não as oferecem nativamente (ainda que existam opções na comunidade) e o Voldemort oferece uma interface básica à parte escrita em Ruby, que está sem manutenção. O Aerospike oferece uma interface que deve ser instalada à parte. No Hazelcast, esta funcionalidade está disponível apenas na versão comercial. O Riak KV é o único onde a interface de gerenciamento já está integrada ao código principal do programa, não requerendo nenhuma instalação extra.

A respeito de ferramentas de monitoramento, o Redis oferece comandos como *INFO*, *MEMORY* e *LATENCY*, enquanto que o Memcached oferece o comando ‘stats’ e o Aerospike oferece o comando ‘asadm’. O Voldemort possui uma interface completa de monitoramento exposta através de Java *Management Extensions* (JMX).

Esta é a mesma estratégia utilizada pelo Hazelcast. O Riak KV oferece os comandos ‘stat’ e ‘stats’ em sua interface de linha de comando (CLI) ‘*Riak KV-admin*’ e a URL ‘/stats’ em sua API HTTP.

Enquanto que Memcached e Hazelcast não oferecem ferramentas próprias para realização de *benchmark*, no Redis existe a ferramenta *redis-benchmark* e o Voldemort oferece a ‘*voldemort-performance-tool*’. No Aerospike os *benchmarks* estão nos clientes disponibilizados e no Riak KV o nome dado ao *benchmark* é Basho Bench.

Após comparar as características dos bancos de dados chave-valor com armazenamento em memória, Redis, Memcached, Voldemort, Aerospike, Hazelcast e Riak KV, é fácil entender o motivo pelo qual o Memcached não é considerado um banco de dados, pois seu foco destoa bastante de seus semelhantes. É possível perceber também que, mesmo que o Redis tenha oferecido suporte à clusterização em suas versões mais recentes, os outros sistemas ainda estão à frente quando o assunto é funcionalidades para execução distribuída. É possível perceber também como Voldemort e Hazelcast se utilizam do ecossistema Java para prover funcionalidades interessantes, e como o *paper* do Dynamo (DECANDIA, HASTORUN, *et al.*, 2007) influencia principalmente Voldemort e Riak KV.

3.3.2 Bancos de dados de famílias de colunas

Esta seção apresenta uma comparação entre as características mercadológicas (Quadro 5), de projeto (Quadro 6) e de manutenção (Quadro 7) dos bancos de famílias de colunas ou *columnar* apresentados na seção 2.5.2.

Ainda que o Cassandra seja o representante mais famoso desta categoria, o Bigtable (CHANG, DEAN, *et al.*, 2008) é considerado o grande influenciador no modelo de dados dos bancos de famílias de colunas. Enquanto que o Cassandra trabalha com uma arquitetura distribuída semelhante ao Dynamo (DECANDIA, HASTORUN, *et al.*, 2007) e um modelo de dados semelhante ao Bigtable, o HBase pode ser considerado uma implementação do Bigtable baseado no Hadoop e o Accumulo é mais integrado com o ecossistema da Apache, fazendo uso de Hadoop, ZooKepper e Thrift.

Quadro 5 - Características mercadológicas dos bancos de famílias de colunas

	Bigtable	**HBase**	**Cassandra**	**Accumulo**
Lançamento	2005	2008	2008	2008
Licença	Proprietária	Apache 2	Apache 2	Apache 2
Desenvolvido	C++, Java, Python, Go, Ruby	Java	Java	Java
SOs Suportados	Google Cloud	JVM	JVM	JVM
Qtde de Linguagens Suportadas	3 oficiais	1 oficial e 4 da comunidade, além das 18 oficiais do Thrift	12 da comunidade	18 oficiais do Thrift
Protocolos	API RCP e API HBase	Thrift e API HTTP REST	Próprio (CQL)	Thrift

O Bigtable se diferencia dos outros bancos estudados por ser oferecido pela Google apenas com licenciamento proprietário e em modelo SaaS, no ambiente de cloud da própria empresa. A implementação do mesmo segue as mesmas linguagens utilizadas em outros sistemas da empresa, com foco majoritário em C++ e Java.

Os bancos HBase, Cassandra e Accumulo, por sua vez, são todos projetos da fundação Apache, distribuídos sob a licença Apache 2. Os mesmos são implementados em Java e rodam na JVM.

A respeito da comunicação com os bancos, o Bigtable oferece clientes em 3 linguagens e uma API baseada em RPC, além de suportar a API HTTP REST do HBase. Tanto HBase como Accumulo oferecem suporte ao framework Thrift, mas o primeiro oferece também uma API HTTP REST própria. O Cassandra possui clientes em várias linguagens e uma linguagem própria chamada Cassandra Query Language ou CQL. O suporte do Cassandra ao framework Thrift foi tornado obsoleto em favor da CQL, portanto ele não é considerado neste estudo.

Quadro 6 - Características do projeto dos bancos de famílias de colunas

	Bigtable	**HBase**	**Cassandra**	**Accumulo**
Escabilidade horizontal	Mestre único	Mestre único	*Shared-nothing*, fator de replicação	Mestre único
Teorema CAP	CP	CP	Configurável	CP
Controle de concorrência	MVCC	MVCC	CAS e Paxos	MVCC
Transações	Não	Não	Sim	Básicas
Garantias da persistência em disco	*Journal*	*Journal*	*Journal* configurável por processo	*Journal* configurável por tabela e por sessão
Failover	Automático	Automático	Automático com *hinted handoff*	Automático
Autenticação e autorização	Google Cloud	SASL, Kerberos	Simples, plugável	SSL, Kerberos

O suporte a escalabilidade ou clusterização é implementada através da arquitetura mestre único no Bigtable, HBase e Accumulo, sendo que o primeiro funciona em cima do Colossus (antigamente chamado de Google File System ou GFS) e os dois últimos dependem do Hadoop Distributed File System (HDFS). O Cassandra segue a arquitetura focada em disponibilidade do Dynamo e funciona com nós idênticos, sem um ponto único de falha e com fator de replicação selecionável por operação.

No espectro do teorema CAP, Bigtable, HBase e Accumulo se classificam como sistemas consistentes (CP) por sua arquitetura de mestre único. Ainda que existam mestres de backup para assumir em caso de falha do mestre, o teorema CAP ressalta que qualquer nó deve ser capaz de responder uma requisição sem dependência de outro nó. O Cassandra, apesar de implementar uma arquitetura focada em disponibilidade (sendo, na configuração padrão, um sistema CA), pode ser configurado para prezar pela consistência, ao custo de redução do desempenho e disponibilidade.

Para o controle de concorrência, o Cassandra utiliza *compare-and-swap* (já explicado na seção 3.3.1) para prover atomicidade e isolamento a nível de célula (duas das garantias ACID) e os nós utilizam o protocolo Paxos para obter alta consistência quando solicitado. Bigtable, HBase e Accumulo usam o método MVCC (também explicado na seção 3.3.1).

Quanto às garantias da persistência em disco, todos os bancos estudados nesta categoria utilizam o método de *journal* (comumente utilizado pelos bancos tradicionais, é um log de operações em arquivo *append-only* também chamado de *Write Ahead Log* ou WAL) para garantir a durabilidade dos dados. Neste ponto, destaca-se a alta granularidade da configuração oferecida pelo Accumulo, pois é possível configurar a sincronização do *journal* para o disco a nível de tabela e até mesmo de sessão. O Cassandra também possui uma opção global que permite configurar a forma como o *fsync* é feito.

O suporte à transações não é oferecido pelo Bigtable e HBase, e atomicidade e isolamento são garantidos apenas para operações em uma única célula. O Accumulo oferece transações básicas, no sentido de que é possível definir condições que devem ser atendidas para que operações sejam executadas, e ambas são executadas com um *lock*. O Cassandra vai além, oferecendo transações (chamadas de *lightweight transactions*) consistentes mesmo entre diferentes nós do cluster, utilizando o protocolo de consenso Paxos (bastante semelhante ao *two-phase commit* utilizado pelos bancos de dados tradicionais).

Todos os bancos oferecem *failover* automático, mas a implementação naturalmente diverge devido às diferenças de arquitetura. Bigtable, HBase e Accumulo, pela sua característica de mestre único, oferecem mestres de backup cuja principal função é assumir a liderança em caso de falha do mestre primário. O mestre primário, por sua vez, controla a distribuição de dados entre os nós do *cluster* e o *failover* dos nós. No Cassandra os nós comunicam-se uns com os outros através do protocolo *gossip* e usa-se *hinted handoff* (em que um nó responde por requisições de responsabilidade de outro nó enquanto este está inoperante e depois repassa as operações realizadas) para garantir a disponibilidade em caso de falha.

Os métodos de autenticação suportados pelos bancos são bastante variados, o Bigtable possui autenticação integrada com o Google Cloud, no chamado *Identity Access Management* (IAM), enquanto que HBase oferece os métodos SASL e integração com Kerberos. O Accumulo também oferece suporte ao Kerberos, mas oferece também autenticação via certificados SSL. Por fim, o Cassandra suporta nativamente autenticação e autorização simples baseadas no próprio banco de dados e oferece suporte a módulos plugáveis de autenticação para estender esta funcionalidade.

Quadro 7 - Características de manutenção dos bancos de famílias de colunas

	Bigtable	**HBase**	**Cassandra**	**Accumulo**
Interface de Gerenciamento	Google Cloud Platform Console	HBase Web UI	Não	Não
Ferramentas de Monitoramento	Stackdriver Monitoring	JMX e JSON na Web UI	JMX	Accumulo Monitor e Hadoop Metrics2 (JMX, Graphite, Ganglia)
Benchmark embutido	Não	LoadTestTool	cassandra-stress	Não

A respeito das interfaces de gerenciamento, enquanto que o Bigtable oferece o painel de gestão do Google (*Google Cloud Platform Console*), o HBase vem com o *HBase Web UI* que permite tanto a gestão quanto o monitoramento dos recursos computacionais (inclusive por uma API HTTP REST, reportando métricas em JSON). Cassandra e Accumulo não oferecem painéis de gestão nativos, ainda que o Accumulo ofereça uma interface visual para monitoramento.

Além do já mencionado HBase Web UI, o HBase, assim como o Cassandra, oferece integração com JMX para monitoramento da JVM. O Bigtable oferece monitoramento via API através do Stackdriver, outra tecnologia que compõe o Google Cloud. Por final, o Accumulo oferece uma interface visual para monitoramento chamada Accumulo Monitor e métricas através do Hadoop Metrics2, que possui integração com JMX, Graphite e Ganglia.

Por sua característica SaaS, não é de surpreender a ausência de um *benchmark* embutido no Bigtable, mas chama atenção esta ausência no Accumulo. Para este fim o Cassandra oferece o *cassandra-stress* e o HBase oferece o *LoadTestTool*.

3.3.3 Bancos de dados orientados a documentos

Esta seção apresenta uma comparação entre as características mercadológicas (Quadro 8), de projeto (Quadro 9) e de manutenção (Quadro 10) dos bancos orientados a documentos apresentados na seção 2.5.3 e do OrientDB que foi apresentado na seção 2.5.4.2, entre os bancos de grafos, mas que também pode ser considerado um banco de documentos.

O MarkLogic é considerado um dos precursores dos bancos NoSQL modernos e tem conseguido se manter dentre os bancos mais populares de sua categoria, porém

o MongoDB é hoje considerado o banco NoSQL mais popular dentre todas as categorias (DB-ENGINES). CouchDB e Couchbase são fortes concorrentes, enquanto que o OrientDB é considerado uma opção mais viável quando há a necessidade de um banco de documento e de um banco de grafos (YUHANNA, LEGANZA e AUSTIN, 2016).

Quadro 8 - Características mercadológicas dos bancos orientados a documentos

	MongoDB	**CouchDB**	**Couchbase**	**MarkLogic**	**OrientDB**
Lançamento	2009	2005	2010	2001	2010
Licença	AGPL e proprietária	Apache 2	Apache 2 e proprietária	Proprietária	Apache 2
Desenvolvido	C++, C e JavaScript	Erlang	C++, C, Erlang e Go	C++ e JavaScript	Java
SOs Suportados	Windows, OS X, Ubuntu, Debian, SUSE, RHEL, Amazon, Linux	Windows, OS X, FreeBSD, Unix-like	Windows, Ubuntu, Debian, Red Hat, SUSE, CentOS	Windows, OS X, Red Hat, SUSE, CentOS, Amazon Linux, Solaris	JVM
Qtde de Linguagens Suportadas	10 oficiais e 32 da comunidade	11 oficiais	9 oficiais e 2 da comunidade	2 oficiais	3 oficiais e 12 da comunidade
Protocolos	Próprio	API HTTP	Memcached	XDBC, API HTTP, SQL (read-only)	Próprio, API HTTP REST

A respeito dos sistemas operacionais, vale ressaltar que MongoDB, Couchbase e MarkLogic não suportam mais sistemas 32 bits, oferecendo apenas instaladores para SO 64 bits em suas versões mais atuais. O suporte ao OS X do MarkLogic é apenas para fins de desenvolvimento, e quanto ao OrientDB, como o mesmo roda na JVM (*Java Virtual Machine*), podem-se considerar os sistemas operacionais suportados por esta tecnologia.

Nota-se o baixo número de linguagens suportadas pelos bancos MarkLogic e OrientDB pela tendência de utilização destes sistemas diretamente através das APIs HTTP REST, API esta que também é oferecida pelo CouchDB.

Quadro 9 - Características do projeto dos bancos orientados a documentos

	MongoDB	**CouchDB**	**Couchbase**	**MarkLogic**	**OrientDB**
Escabilidade horizontal	Mestre único, fator de replicação	*Shared-nothing*, fator de replicação	*Shared-nothing*, fator de replicação	Mestre-mestre	Mestre-mestre e quórum
Teorema CAP	CP	AP	CP	CP	AP
Controle de concorrência	MGL	MVCC	*Compare-and-swap*	MVCC	MVCC
Transações	Não	Não	Parcial	Sim	Sim
Garantias da persistência em disco	*Journal* configurável por operação	*Append-only* configurável	Padrão é assíncrona, configurável por operação	*Journal* configurável por banco	*Journal* configurável por processo
Failover	Automático	Manual	Automático	Automático	Automático
Autenticação e autorização	Própria e SSL. Na comercial, Kerberos e LDAP	Própria, simples, OAuth	SASL, LDAP e PAM	Própria, LDAP e Kerberos	Própria e SSL

No MongoDB a escalabilidade é oferecida através do método conhecido como mestre único, onde todas as consultas e escritas são encaminhadas pelos nós para o mestre. Também é possível configurar para que seja possível ler dados das réplicas, diminuindo o nível de consistência da consulta. CouchDB e Couchbase oferecem uma arquitetura *shared-nothing*, onde cada nó é igual aos outros, com fator de replicação configurável. O MarkLogic implementa escalabilidade pela arquitetura *multi-master* ou mestre-mestre, onde existem vários servidores que atendem como mestres. O OrientDB também utiliza-se desta arquitetura, ainda aplicando o mecanismo de quórum, onde pelo menos um determinado quórum de mestres devem processar o comando com sucesso para que o mesmo seja aceito.

Dentro do teorema CAP explicado no item 2.1.12, MongoDB, Couchbase e MarkLogic são considerados sistemas CP enquanto que CouchDB e OrientDB são sistemas AP, quando configurados para tal. Todos eles oferecem configurações que violam algumas restrições do teorema, tais como configurar o OrientDB com *write quorum* maior que 1, o que reduz a disponibilidade em favor da consistência; ou configurar o Couchbase para permitir que nós de replicação respondam a leituras, aumentando a disponibilidade e reduzindo a consistência. Estas configurações, porém, não fazem com que o sistema mude de categoria dentro do teorema CAP.

Quanto ao controle de concorrência, o MongoDB aplica MGL (*Multiple Granularity Locking* ou bloqueio de granularidade múltipla), ou seja, o *lock* é feito no menor nível (global, banco de dados, coleção ou documento) possível para garantir a consistência de operações concorrentes. Todos os outros bancos utilizam métodos de bloqueio otimista (*optimistic locking*): o Couchbase usa CAS (*compare-and-swap*) e CouchDB, MarkLogic e OrientDB utilizam MVCC, ambos explicados na seção 3.3.1.

Enquanto MongoDB e CouchDB não oferecem nenhum suporte à transações, Couchbase oferece suporte apenas a transações que envolvam um único documento e sugere em sua documentação que o cliente implemente uma forma de *two-phase commit* caso seja necessário. Tanto MarkLogic quando OrientDB oferecem transações mesmo em ambientes distribuídos, mas enquanto que o primeiro utiliza uma forma de *two-phase commit* onde a transação acontece simultaneamente nos diferentes nós do cluster, o segundo aplica uma validação de quórum, onde a transação é aceita quando a maioria dos nós entende que a transação ocorreu de forma satisfatória e o restante dos nós é sincronizado posteriormente.

A respeito das garantias da persistência em disco, o único banco que não se utiliza da técnica de *journal* (explicada na seção 3.3.2) é o Couchbase, fazendo a gravação em disco de forma assíncrona por padrão, porém permitindo a configuração de sincronicidade a nível de operação, para favorecer a consistência em favor do desempenho. O CouchDB se destaca por gravar o próprio banco de dados de forma *append-only*, diferentemente dos outros bancos que gravam um *journal* que é constantemente reescrito por uma operação de *background*. Nota-se, porém, que o CouchDB não oferece uma forma nativa de reescrita de seu arquivo de banco, portanto o mesmo cresce em um ritmo constante. O nível de configuração de sincronicidade oferecida pelos bancos no *journal* também é bastante variável, sendo por operação no MongoDB, por banco de dados no MarkLogic e por processo (instância do banco) no OrientDB (a escrita é considerada síncrona quando ocorre antes do retorno ao cliente, e assíncrona quando o sistema não espera a escrita do *journal* para dar retorno ao cliente).

O único banco que não oferece a opção de *failover* automático é o CouchDB, enquanto que o restante se baseia no esquema de *heartbeats* (pequenos pacotes de dados enviados entre os nós do cluster em intervalos pré-configurados que definem se os nós estão disponíveis). Nos bancos baseados em nós mestres, como MongoDB,

MarkLogic e OrientDB, quando algum mestre cai os nós de replicação promovem uma eleição no cluster para se autopromoverem a mestres.

Quanto a autenticação e autorização, ambas são suportadas por todos os bancos avaliados, e apenas o Couchbase não oferece uma API própria para gerenciamento de usuários dentro do próprio banco de dados. Autenticação baseada em certificados SSL é suportada pelo MongoDB e pelo OrientDB. Tanto MarkLogic quanto a versão comercial do MongoDB suportam autenticação pelo Kerberos e pelo protocolo LDAP, sendo que esta última também é oferecida pelo Couchbase.

Quadro 10 - Características de manutenção dos bancos orientados a documentos

	MongoDB	**CouchDB**	**Couchbase**	**MarkLogic**	**OrientDB**
Interface de Gerenciamento	Apenas comercial	Sim	Sim	Sim	Sim
Ferramentas de Monitoramento	Comandos e ferramenta comercial	URL REST	Na interface e URL REST	Na interface e URL REST	JMX
Benchmark embutido	Sim	Não	Não	Não	Não

O MongoDB é o único dos bancos de documentos avaliados que não oferece uma interface de gerenciamento gratuita e embutida no sistema (ainda que existam várias criadas pela comunidade). Em todos os outros bancos existe uma interface de gerenciamento web que permite efetuar atividades de gerenciamento e monitoria dos mesmos.

Couchbase e MarkLogic se destacam por oferecerem métricas de monitoramento, incluindo históricos, na própria interface de gerenciamento da ferramenta, além de oferecerem *web services* REST com estas métricas. O CouchDB também oferece um *web service*, enquanto que o OrientDB expõe métodos de monitoramento através do JMX e o MongoDB oferece comandos internos e uma ferramenta comercial para monitoramento.

Por final, o MongoDB é o único banco a oferecer uma ferramenta de *benchmark* embutido no sistema, ainda que o OrientDB ofereça suporte à ferramenta de testes do framework *TinkerPop*.

3.3.4 Bancos de dados de grafos ou triplos

Esta seção apresenta uma comparação entre as características mercadológicas (Quadro 11), de projeto (Quadro 12) e de manutenção (Quadro 13) dos bancos de grafos ou triplos, apresentadas na seção 2.5.4.

O Neo4j é o banco de grafos mais popular atualmente e o OrientDB já foi apresentado anteriormente, na seção 3.3.3, dentre os bancos de documentos, por ser um banco multi-modelo. O JanusGraph surgiu a partir do Titan, encerrado em 2015, e o Graph Engine era conhecido como Trinity em suas primeiras versões. O Bitsy, por fim, é um banco de grafos com arquitetura *serverless*, que roda embutido na aplicação cliente.

Quadro 11 - Características mercadológicas dos bancos de grafos ou triplos

	Neo4j	**OrientDB**	**JanusGraph**	**Graph Engine**	**Bitsy**
Lançamento	2007	2010	2017	2017	2013
Licença	GPLv3 e proprietária	Apache 2	Apache 2	MIT	Apache 2
Desenvolvido	Java e Scala	Java	Java	C# e C++	Java
SO Suportados	JVM	JVM	JVM	Windows e Ubuntu	JVM
Qtde de Linguagens Suportadas	4 oficiais	3 oficiais e 12 da comunidade	8 oficiais do TinkerPop	1 oficial	1 oficial
Protocolos	Próprio (Bolt), API HTTP REST	Próprio, API HTTP REST	Próprio e TinkerPop	API HTTP REST	N/A

Enquanto que o Neo4j destaca-se por sua maturidade e presença de mercado, o OrientDB surge como um forte concorrente. Quanto ao JanusGraph, apesar de seu pouco tempo de mercado, deve-se considerar que ele nasceu de um *fork* no código-fonte do Titan, que foi ativamente desenvolvido entre 2012 e 2015. O Graph Engine surge como uma boa alternativa para os adeptos do ecossistema da Microsoft, e o Bitsy destoa bastante de seus concorrentes pelo seu foco *serverless*.

O modelo de licenciamento do Neo4j é mais restritivo do que seus concorrentes, pois obriga a compra de uma licença proprietária para a distribuição de softwares comerciais. Nota-se a grande adoção do Java como linguagem de desenvolvimento e a consequente utilização da JVM como plataforma de execução, com exceção do Graph Engine, que roda na plataforma do .NET Framework.

A respeito das linguagens suportadas, vale destacar que o Graph Engine suporta apenas as linguagens do .NET, e que o Bitsy oferece suporte apenas ao Java, enquanto que o JanusGraph é completamente compatível com o framework *TinkerPop* e, portanto, suporta as 8 linguagens oficiais listadas no site deste framework.

Excetuando-se o Bitsy, que roda embutido e, portanto, não utiliza protocolos de comunicação via *socket*, todos os bancos de grafos estudados oferecem uma API HTTP REST (no JanusGraph, esta API é representada pelo componente Rexter do *TinkerPop*).

Quadro 12 - Características do projeto dos bancos de grafos ou triplos

	Neo4j	**OrientDB**	**JanusGraph**	**Graph Engine**	**Bitsy**
Escabilidade horizontal	2 formas, mas apenas na versão comercial	Mestre-mestre e quórum	Depende do *storage*	Mestre - Escravo	Não
Teorema CAP	CP	AP	Depende do *storage*	-	N/A
Controle de concorrência	*Lock* tradicional	MVCC	Depende do *storage*	*Spinlock* por registro	*Optimistic locking*
Transações	Sim	Sim	Sim	Sim	Sim
Garantias da persistência em disco	*Journal* síncrono	*Journal* configurável por processo	Depende do *storage*	*Journal* configurável por operação	*Append-only* síncrono
Failover	Apenas comercial, com quórum	Automático	Depende do *storage*	Automático, com *heartbeats*	N/A
Autenticação	Simples. Na comercial, LDAP e autorização	Própria e SSL	SSL e SASL do TinkerPop	Não	Não

Por seu foco mais comercial, o Neo4j oferece duas opções de escalabilidade horizontal, mas apenas em sua versão paga: *causal cluster*, onde existem múltiplos mestres e é utilizado um fator de replicação fixo; e *highly available cluster*, onde existe um único mestre e os outros nós do cluster são réplicas, mesmo método utilizado pelo MongoDB, conforme explicado na seção 3.3.3. Nenhuma das duas formas oferece suporte a *sharding*, portanto todos os nós contêm todos os dados, enquanto que o OrientDB suporta *sharding* e sua implementação de cluster está explicada na seção 3.3.3.

O Bitsy não possui suporte à escalabilidade horizontal, por seu foco *serverless*, e o JanusGraph não suporta escalabilidade nativamente, delegando ao módulo de armazenamento o provimento desta funcionalidade. O Graph Engine implementa *sharding* através da divisão da memória RAM das máquinas em blocos de memória (*memory trunks*), e o método mestre-escravo para coordenação do cluster, onde o mestre mantêm um registro da localização de todos os dados dentro do cluster.

Quanto à abordagem ao teorema CAP explicado no item 2.1.12, o mesmo não se aplica ao Bitsy por este não prover escalabilidade horizontal, o Neo4j prioriza a consistência frente a partições (CP) em ambas as opções de clusterização, e o OrientDB prioriza a disponibilidade (AP), conforme explicado na seção 3.3.3. Dentre os *storage backends* oferecidos pelo JanusGraph, o Cassandra preza pela disponibilidade (AP) e o HBase preza pela consistência (CP), enquanto que o BerkeleyDB não suporta escalabilidade horizontal. O Graph Engine não cumpre os requisitos de nenhuma das facetas do teorema CAP, pois não provê a linearizabilidade (*linearizability*) necessária para a classificação como CP devido à sua execução paralelizada, e nem a dissociação entre nós necessária para a classificação como AP (impossível em um esquema mestre - escravo, haja visto que o escravo depende do mestre para prover uma resposta).

Enquanto que o OrientDB utiliza-se do MVCC para controlar a concorrência, o Neo4j faz bloqueios (*locks*) de leitura (*read lock*, não exclusivos, onde múltiplas transações podem obter o mesmo *lock*) e de escrita (*write lock*, exclusivos, onde apenas uma transação pode obter o *lock*). O JanusGraph não faz nenhum controle adicional sobre a concorrência, deixando isso a cargo da camada de armazenamento. O Bitsy utiliza-se do método de bloqueio otimista, onde não são necessários *locks* perceptíveis pelo usuário e em caso de operações concorrentes a segunda falha. O controle de concorrência do Graph Engine baseia-se em *spinlocks* a nível de registro ou célula.

Todos os bancos estudados nesta categoria oferecem suporte à transações e oferecem garantias ACID, exceto o Graph Engine, que não provê serializabilidade (*serializability*) para *threads* concorrentes. Nota-se que as garantias ACID do JanusGraph mais uma vez dependem da camada de armazenamento escolhida.

Quanto às garantias da persistência em disco, tanto Neo4j quanto Graph Engine utilizam *jornal* (explicado na seção 3.3.2), porém no Neo4j ele é síncrono e no

Graph Engine a sincronicidade pode ser configurada por operação. O Bitsy utiliza uma técnica semelhante ao CouchDB, estudado na seção 3.3.3, onde o próprio banco é escrito na forma de um arquivo *append-only* no disco, porém neste caso a sincronização com o disco é sempre síncrona. Esta característica também depende do *storage backend* no JanusGraph.

No Neo4j o *failover* é oferecido de forma automatizada através de quórum entre os mestres, porém apenas na versão comercial. No Graph Engine o *failover* é automático através de *heartbeats* entre os nós do cluster, enquanto que no JanusGraph ele depende da camada de armazenamento e no Bitsy esta característica não se aplica.

A autenticação não é oferecida pelo Bitsy e pelo Graph Engine, e o JanusGraph oferece suporte aos protocolos SSL e SASL através do framework *TinkerPop*. Na versão comunitária do Neo4j é oferecida apenas autenticação simples, enquanto que na versão comercial há integração com LDAP e suporte completo à autorização. O OrientDB já foi explanado na seção 3.3.3.

Quadro 13 - Características de manutenção dos bancos de grafos ou triplos

	Neo4j	**OrientDB**	**JanusGraph**	**Graph Engine**	**Bitsy**
Interface de Gerenciamento	Sim	Sim	The Dog House, da TinkerPop	Não	Não
Ferramentas de Monitoramento	Na comercial, suporte ao Graphite e exportação para CSV	JMX	Suporte a Graphite, CSV, JMX e outros	Não	JMX
Benchmark embutido	Não	Não	Não	Não	Não

O Neo4j oferece a interface de gerenciamento Neo4j Browser, enquanto que o JanusGraph não oferece nenhuma interface própria, mas pode-se utilizar a *The Dog House*, do framework *TinkerPop*. O Bitsy e o Graph Engine não oferecem interface de gerenciamento. Estas interfaces oferecem suporte não só à configuração e gerenciamento do servidor/cluster, como também execução de consultas e visualização de dados do grafo.

Assim como o OrientDB, o Bitsy e o JanusGraph oferecem suporte ao protocolo JMX para monitoramento e o JanusGraph ainda oferece suporte a logs no console, geração de arquivos CSV, e integração com Ganglia, Graphite e Slf4j, além da possibilidade de plugar módulos próprios de monitoramento. O Neo4j oferece suporte

ao Graphite e exportação de arquivos CSV. O Graph Engine é o único que não oferece formas de monitoramento do cluster e seus nós.

Nenhum dos bancos estudados oferece um benchmark embutido, ainda que exista a ferramenta do framework *TinkerPop* que pode ser aplicada em todos eles, exceto no Graph Engine.

3.4 SELEÇÃO DO BENCHMARK E BANCOS DE DADOS PARA TESTES

Devido a ampla utilização e ao suporte à maioria dos bancos de dados apresentados anteriormente nessa pesquisa, optou-se por utilizar o YCSB como *benchmark* para execução das cargas de trabalho.

O YCSB possui seis cargas padrões identificadas por letras de A até F, além de possibilitar a personalização ou criação de novas cargas utilizando um *template* distribuído junto com estas cargas. A personalização ou modificação das cargas é bastante simples, pois as mesmas são definidas em arquivos de texto simples, com pares de propriedades e valores.

O Quadro 14 apresenta as cargas de trabalho disponibilizadas, além de uma carga personalizada criada pelos autores deste trabalho. Todas as cargas trabalham com registros de 1KB, compostos por 10 campos com 100 bytes cada, mais o tamanho da chave.

Quadro 14 - Cargas de trabalho YCSB

Workload	Foco e resumo	Caso de Uso	Requisições	Distribuição das solicitações
A	Atualização (*update*)	Armazenamento de sessão que registra ações recentes.	50% leitura e 50% alteração	Zipfian
B	Principalmente leitura	Etiquetagem de fotos (*tags*); Adicionar uma *tag* é uma atualização, mas a maioria das operações são para ler *tags*.	95% leitura e 5% alteração	Zipfian
C	Somente leitura	Cache do perfil do usuário, onde os perfis são construídos em outro lugar.	100% leitura	Zipfian
D	Leitura dos últimos (*read latest*) Novos registros são inseridos, e os mais recentes são lidos.	Atualizações de status do usuário e leitura de notícias.	95% leitura e 5% inserção	Últimos (*latest*)
E	Intervalos curtos (*short ranges*) São consultados intervalos curtos de registros, em vez de registros individuais.	Comentários ou conversas contextualizadas, onde a busca é pelas conversas em determinado contexto.	95% s*can* e 5% inserção	Zipfian
F	Fluxo leitura-modificação-escrita (*read-modify-write*) Um registro é lido, modificado, e gravado de volta.	Dados de usuários, onde o usuário insere, lê e altera os registros, ou suas atividades são registradas.	50% leitura e 50% leitura-alteração-escrita	Zipfian
Personalizado	Inserção e leitura	Cadastro e consulta de dados com pouca alteração, tais como notas fiscais eletrônicas	50% inserção e 50% leitura	Zipfian
Personalizado 2	Inserção e leitura	Cadastro e consulta de dados com pouca alteração, tais como notas fiscais eletrônicas	75% inserção e 25% leitura	Zipfian

FONTE: Adaptado de (YAHOO, 2011)

Na última linha do Quadro 14 é possível observar o *workload* “Personalizado” utilizado nos testes do presente estudo para os bancos de dados chave-valor. Os testes foram executados com uma carga de aproximadamente 5 GB, compostos por 5 milhões de operações de 1 KB cada, distribuídos em 2,5 GB de inserção e 2,5 GB de leitura de dados.

Para os bancos de dados orientados a documentos e para o Cassandra a carga utilizada foi a “Personalizado 2”, onde foram executadas operações sobre uma carga de 10GB, composta por 10 milhões de operações de 1KB, sendo 7,5GB aproximadamente de inserção e 2,5GB de leitura.

Cada teste foi repetido 30 vezes, executando-se também a etapa de limpeza e preparação do banco entre cada uma das execuções. Foram testados 2 sistemas de cada uma das categorias de banco de dados NoSQL estudadas, exceto os bancos de grafos, pois o YCSB não oferece suporte a bancos desta categoria.

A saída do *benchmark* já oferece várias métricas, conforme pode ser visualizado no Quadro 15, com base nessas saídas foram realizados os testes estatísticos.

Quadro 15 - Saída do *benchmark*

Operação	**Métrica**	**Resultado**
[OVERALL]	*RunTime(ms)*	4881866,0
[OVERALL]	*Throughput(ops/sec)*	2048,397067842501
[TOTAL_GCS_PS_Scavenge]	*Count*	4189,0
[TOTAL_GC_TIME_PS_Scavenge]	*Time(ms)*	28907,0
[TOTAL_GC_TIME_%_PS_Scavenge]	*Time(%)*	0,5921301404012318
[TOTAL_GCS_PS_MarkSweep]	*Count*	8,0
[TOTAL_GC_TIME_PS_MarkSweep]	*Time(ms)*	1301,0
[TOTAL_GC_TIME_%_PS_MarkSweep]	*Time(%)*	0,026649645852630938
[TOTAL_GCs]	*Count*	4197,0
[TOTAL_GC_TIME]	*Time(ms)*	30208,0
[TOTAL_GC_TIME_%]	*Time(%)*	0,6187797862538628
[READ]	*Operations*	2500839,0
[READ]	*AverageLatency(us)*	5981,679684297949
[READ]	*MinLatency(us)*	187,0
[READ]	*MaxLatency(us)*	4988927,0
[READ]	*95thPercentileLatency(us)*	31407,0
[READ]	*99thPercentileLatency(us)*	122623,0
[READ]	*Return=OK*	2500839
[CLEANUP]	*Operations*	4,0
[CLEANUP]	*AverageLatency(us)*	24,0
[CLEANUP]	*MinLatency(us)*	7,0
[CLEANUP]	*MaxLatency(us)*	60,0
[CLEANUP]	*95thPercentileLatency(us)*	60,0
[CLEANUP]	*99thPercentileLatency(us)*	60,0
[INSERT]	*Operations*	7499161,0
[INSERT]	*AverageLatency(us)*	501,8017304869171
[INSERT]	*MinLatency(us)*	217,0
[INSERT]	*MaxLatency(us)*	9945087,0
[INSERT]	*95thPercentileLatency(us)*	815,0
[INSERT]	*99thPercentileLatency(us)*	2589,0
[INSERT]	*Return=OK*	7499161
[OVERALL]	*RunTime(ms)*	4881866,0

3.4.1 Bancos de dados chave-valor

Os bancos de dados chave-valor que se destacaram durante a análise comparativa e que, portanto, foram selecionados para o teste de carga foram Redis versão 3.2.8 e Aerospike *Community Edition* versão 3.12.1.1.

O Redis é considerado o sistema mais popular desta categoria (DB-ENGINES), sendo largamente utilizado na indústria e muito estudado na academia. Ele também é citado como *challenger* na análise dos quadrantes mágicos da Gartner de 2016 (GARTNER, 2016), tendo perdido a categoria de *leader* que ocupou no ano anterior, e como *leader* na análise de bancos NoSQL da Forrester (YUHANNA, LEGANZA e AUSTIN, 2016).

O Aerospike é apontado como *strong performer* nas análises da Forrester a respeito de bancos de dados NoSQL (YUHANNA, LEGANZA e AUSTIN, 2016) e a respeito de *in-memory database*. Nas análises dos quadrantes mágicos da Garner o Aerospike ocupou a posição de *visionarie* em 2014 e de *niche player* em 2015, tendo sido removido da análise de 2016 por "não preencher os requisitos de receita para inclusão" (GARTNER, 2016). Vale notar que os requisitos de receita da análise da Gartner foram aumentados de US$ 20 milhões para US$ 50 milhões neste último ano.

3.4.2 Bancos de dados orientados a documentos

Os bancos de dados orientados a documentos que se destacaram durante a análise comparativa e que, portanto, foram selecionados para o teste de carga foram MongoDB versão 3.4.4 e Couchbase versão 4.5.1.

Além de ser considerado o banco mais famoso dentre todas as categorias de bancos NoSQL, o MongoDB é citado como *challenger* na análise dos quadrantes mágicos da Gartner de 2016 (GARTNER, 2016), já tendo sido apontado como *leader* em 2015. Na análise de bancos NoSQL voltados a BigData da Forrester, o MongoDB é citado com principal *leader*, acima de concorrentes de peso como Amazon, Oracle e Google (YUHANNA, LEGANZA e AUSTIN, 2016). A análise da Forrester de bancos de documentos também aponta o MongoDB como principal representante da categoria (YUHANNA, LEGANZA e WARRIER, 2016).

O Couchbase também aparece como um *leader* nas análises da Forrester de bancos de documentos (YUHANNA, LEGANZA e WARRIER, 2016) e de bancos NoSQL voltados a BigData (YUHANNA, LEGANZA e AUSTIN, 2016). Na análise da Gartner ele aparece como *niche player* (GARTNER, 2016).

3.4.3 Cassandra

O Cassandra é considerado o banco mais popular dentre os bancos de dados de famílias de colunas, sendo que a empresa DataStax oferece uma versão do mesmo com suporte comercial e, portanto, é ela que aparece nos relatórios da Forrester e da Gartner.

A DataStax aparece como *challenger* nos quadrantes mágicos da Gartner de 2016 (GARTNER, 2016), já tendo sido elencada como *leader* na análise do ano anterior (GARTNER, 2015). Na análise de bancos NoSQL voltados a BigData da Forrester a DataStax também é apontada como *leader* (YUHANNA, LEGANZA e AUSTIN, 2016).

Apesar de terem sido analisados outros bancos de famílias de colunas, os motivos para a exclusividade do Cassandra nos testes para avaliação estão explanados no item 3.6.3.

3.5 PLANEJAMENTO DOS EXPERIMENTOS

3.5.1 Ambiente

O ambiente de infraestrutura físico onde foram realizados os experimentos do presente trabalho é composto por três computadores HP ProLiant DL385 G6, cada um com 2 processadores AMD Opteron 2425HE (6 núcleos operando a 2.1 GHz, com 6 MB de cache L3), 8 pentes de 4 GB de memória RAM DDR2 (*clock* de 800 MHz) e 1 TB de capacidade de armazenamento sob a gerência do controlador de RAID HP Smart Array G6. Os computadores estão ligados por uma rede gigabit e a plataforma OpenNebula é utilizada para gerenciamento do ambiente virtual.

Para os testes foi criada uma máquina virtual para atuar como cliente, onde foi instalado o YCSB versão 0.12.0 seguindo as orientações do repositório oficial em https://github.com/brianfrankcooper/YCSB/, e uma máquina virtual para cada banco de dados, onde foi instalado o banco a ser testado. A máquina cliente possui Ubuntu Server 16.04, 8 GB de memória RAM e 8 núcleos de CPU, além de 10 GB de espaço em disco.

As máquinas dos bancos de dados possuem Ubuntu Server 16.04, 16 GB de memória RAM e 8 núcleos de CPU, além de 20 GB de espaço em disco e 4 GB de

swap, para os bancos de dados chave-valor. E possuem Ubuntu Server 16.04, 8 GB de memória RAM e 8 núcleos de CPU, além de 20 GB de espaço em disco e 4 GB de *swap*, para os bancos de dados orientados a documentos.

Durante a etapa de preparação das máquinas virtuais para os bancos de dados, foram feitas algumas configurações seguindo orientações encontradas nos manuais dos sistemas sendo testados. Foram elas: inativação do *Transparent Huge Pages* (THP), aumento do número limite de *Open File Handles* (*file descriptors*), alteração da *overcommit memory* (*vm.overcommit_memory*) para 1 (*always overcommit*) e alteração da fila de conexões TCP (*net.core.somaxconn*) para 4096.

Também foi configurado um compartilhamento via *Network File System* (NFS) na máquina cliente (que funcionou como servidora do NFS) como forma de transferência de arquivos entre as máquinas virtuais.

3.5.2 Monitoramento

Para automatizar a tarefa de monitoramento e obter dados confiáveis, foram desenvolvidos dois programas simples em linguagem C++: *client*, para rodar na máquina cliente, e *server*, para rodar na máquina do banco de dados.

O programa *server* fica monitorando a pasta compartilhada por NFS aguardando pela criação de um arquivo com o nome “*start*”, para então iniciar a coleta das métricas de desempenho do sistema e do banco de dados, até que seja criado o um arquivo com o nome “*stop*” na mesma pasta. As métricas do sistema são coletadas através de chamadas aos comandos “*iostat*”, “*free*” e “*df*”, e as métricas dos bancos de dados são coletadas através de comandos oferecidos pelo próprio banco, tais como “*redis-cli info all*” para o Redis. Os dados obtidos são gravados na mesma pasta compartilhada.

O programa *client*, por sua vez, lê os conteúdos de uma pasta procurando por arquivos de texto com os comandos do benchmark a ser executado. Quando encontrado o arquivo, o programa cria o arquivo “*start*” na pasta compartilhada por NFS e inicia a execução do benchmark, armazenando a saída deste em um arquivo.

Quando o benchmark é finalizado, o *client* solicita o encerramento do monitoramento através da criação do arquivo “*stop*” na pasta compartilhada e copia os dados obtidos pelo monitoramento do servidor para outra pasta, adicionando

também o arquivo com a saída do *benchmark*. O programa recebe por parâmetro também um valor numérico que identifica quantas vezes este processo deve ser automaticamente repetido.

O CÓDIGO FONTE COMPLETO DOS PROGRAMAS DE MONITORAMENTO PODE SER ESTUDADO NO APÊNDICE C – ARTIGO ESCOLA REGIONAL DE BANCO DE DADOS

Estudo Comparativo de Banco de DADOS CHAVE-VALOR COM Armazenamento em Memória

Dinei A. Rockenbach[1], NADINE ANDERLE[1], DALVAN GRIEBLER[1 2], SAMUEL SOUZA[1]

[1]**LABORATÓRIO DE PESQUISAS AVANÇADAS PARA COMPUTAÇÃO EM NUVEM (LARCC)**

Faculdade Três de Maio (SETREM) – Três de Maio – RS – Brasil

[2]**PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO GRANDE DO SUL (PUCRS/PPGCC)**

Porto Alegre – RS – Brasil

{dineiar,nadianderle}@gmail.com,dalvan.griebler@acad.pucrs.br,

samuel@samuelsouza.com

Abstract. *KEY-VALUE DATABASES EMERGE TO ADDRESS RELATIONAL DATABASES' LIMITATIONS AND WITH the increasing capacity of RAM memory it is possible to offer greater performance and versatility in data storage and processing. The objective is to perform a comparative study of key-value databases with memory storage Redis, Memcached, Volde**MORT, AEROSPIKE, HAZELCAST AND RIAK KV. THUS,** the work contributed to an analysis of different databases and with results that qualitatively demonstrated the characteristics and pointed out the main advantages.*

Resumo. *BANCOS DE DADOS (BD) CHAVE-VALOR SURGEM PARA SUPRIR LIMITAÇÕES DE BDs relacionais e com o aumento da capacidade das memórias RAM é possível oferecer maior desempenho e versatilidade no armazenamento e processamento dos dados. O objetivo é realizar um estudo comparativo dos BDs chave-valor co**M ARMAZENAMENTO EM MEMÓRIA REDIS, MEMCACHED, VOLDEMORT, AEROSPIKE,** Hazelcast e Riak KV. Assim, o trabalho contribuiu para uma análise de diferentes BDs e com resultados que demonstraram qualitativamente as características e apontaram as principais vantagen**S.***

1. Introdução

As necessidades de alta disponibilidade e velocidade, bem como o aumento e consumo **DE DADOS** nos sistemas e aplicações atuais, influenciaram no desenvolvimento de novas **FORMAS PARA O** armazenamento de dados. Estas vão além dos bancos de dado**S RELACIONAIS, IMPULSIONANDO O** movimento NoSQL (***NOTONLYSQL*****)[FOWLER 2015]. NÃO OBSTANTE, COM A POPULARIDADE EM ALTA** em um mercado sem um líder estabelecido, há uma consequente **EXPLOSÃO NO NÚMERO DE**

sistemas de armazenamento disponíveis, o que dificulta a **TOMADA DE DECISÃO** quanto à opção que melhor supre as necessidades organizacionais.

Com o objetivo de solucionar problemas específicos, os bancos NoSQL foram categorizados de acordo com suas características e otimizações. Por exemplo, a Amazon utiliza seu **SISTEMA CHAVE-VALOR (KEY-VALUE) DYNAMO [DECANDIA ET AL. 2007] PARA GERENCIAR AS LISTAS** de mais vendidos, carrinhos de compras, preferências do consumidor, gerenciamento de produtos, entre outras aplicações. Exitem também sistemas de família de colunas (col**UMN** family ou columnar) que foram influenciados pelo Bigtable da Google [Chang et al. 2008]. Enquanto isso, os assim chamados de sistemas de documentos (document) resultaram, por exemplo, no MongoDB. Por fim, do chamado banco triplo **(*GRAPH DATABASE* OU *TRIPLE*), TEM-SE** como exemplo o Neo4j**. CADA UMA DESTAS CATEGORIAS TRAZ SISTEMAS QUE COBREM DIFERENTES** limitações dos bancos relacionais tradicionais.

Este artigo está organizado em 5 seções, incluindo esta seção introdutória. Na Seção 2 estão os trabalhos relac**IONADOS. A SEÇÃO 3 TRAZ O EMBASAMENTO SOBRE OS BANCOS DE DADOS** pesquisados. Na Seção 4 está detalhado o estudo comparativo destes sistemas. Por fim, na Seção 5 estão as conclusões e propostas para trabalhos futuros.

2. Trabalhos Relacionados

Nesta seção é apr**ESENTADA UMA DISCUSSÃO SOBRE OS TRABALHOS RELACIONADOS PUBLICADOS** recentemente na literatura. De forma semelhante, todos os trabalhos escolhidos buscam caracterizar e comparar bancos de dados NoSQL. No entanto, não voltam os estudos para um determinado tip**O DE BANCO DE DADOS, O QUE É IMPORTANTE PARA AVALIAR CARACTERÍSTICAS ESPECÍFICAS.** Tanto [Hecht and Jablonski 2011] quanto [Han et al. 2011] se propõem a fazer um estudo e avaliação dos bancos de dados NoSQL, com o mesmo objetivo principal: prover informaçõ**ES** para auxiliar na escolha do banco NoSQL que melhor atende às necessidades. Por não delimitarem um tipo específico de banco NoSQL, ambos possuem um escopo mais abrangente do que o presente trabalho. No ponto de intersecção entre este trabalho e os citado**S, [HECHT AND** Jablonski 2011] inclui em seu trabalho os bancos Project Voldemort, Redis e Membase, enquanto que [Han et al. 2011] avalia Redis, Tokyo Cabinet-Tokyo Tyrant e Flare.

Em [Deka 2014] é apresentada uma visão geral de vários sistemas NoSQL e os r**EPRESENTANTES CHAVE-VALOR INCLUÍDOS NA AVALIAÇÃO SÃO HYPERTABLE, VOLDEMORT,** Dynomite, Redis e Dynamo. Nota-se a falta, porém, de uma visão comparativa mais clara sobre aspectos de garantias de durabilidade, disponibilidade, protocolos suportados, e outras informações que podem vir a ter uma influência significativa na escolha de um banco de dados chave-valor. Já [Zhang et al. 2015] traz uma visão bem estruturada dos objetivos que nortearam o projeto de cada um dos sistemas descritos no trabalho, o qual foca **EM SISTEMAS COM** gerenciamento e processamento de dados em memória. Dentre os sistemas estudados, os representantes dos bancos chave-valor são MemepiC, RAMCloud, Redis, Memcached, MemC3 e TxCache. Dos sistemas, o trabalho descreve as cargas de dados mais a**DEQUADAS AO SISTEMA, A** estratégia para construção de índices, o controle de concorrência, tolerância a falhas, tratamentos para conjuntos de dados maiores do que a memória disponível e o suporte a consultas personalizadas em baixo nível (como stored proced**URES E SCRIPTS EM LINGUAGEM NATIVA,** por exemplo), porém com pouca abordagem de alto nível que auxilie na escolha de um sistema em favor de outro**.**

Ainda que o tema NoSQL tenha sido bastante explorado na academia, e a bibliografia focada no armazenamento de **DADOS EM MEMÓRIA TENHA CRESCIDO MUITO NOS ÚLTIMOS ANOS, NOTA-**se a falta de um estudo comparativo entre os sistemas chave-valor em memória Redis, Memcached, Voldemort, Aerospike, Hazelcast e Riak KV.

3. Banco de Dados com Armazenamento em Memória

O crescimento DOS BANCOS DE DADOS COM ARMAZENAMENTO DE DADOS EM MEMÓRIA (IMDB OU IN-memory databases) segue uma tendência que teve seu início no hardware, com a capacidade da memória dobrando em média a cada três anos e seu preço caindo uma casa decimal a cada cinco an OS [ZHANG ET AL. 2015]. AS MEMÓRIAS NÃO-VOLÁTEIS (NVM OU NON-VOLATILE MEMORY) COMO o SSD (Solid State Disk), também têm evoluído, mas seu custo [Kasavajhala 2011], durabilidade e confiabilidade [Schroeder et al. 2016] continuam sendo impeditivos para a mai ORIA DAS APLICAÇÕES.

A vantagem em manter os dados na memória ao invés do disco está relacionada à latência de acesso a estes dados, pois remove-se a necessidade de acessar a camada mais lenta da hierarquia de memória, conforme demonstrado pela Figura 1 (a DAPTADA DE [ZHANG ET AL. 2015]), que detalha as camadas de arm AZENAMENTO, BEM COMO UMA ESTIMATIVA DE SUA capacidade atual e da latência de acesso aos dados nela armazenados.

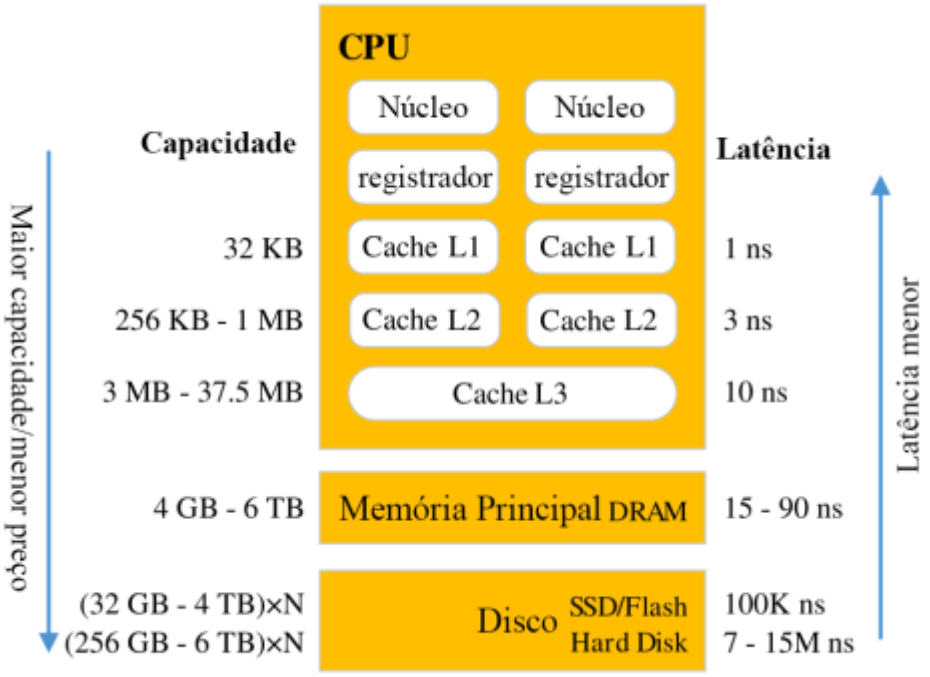


Figura 1. Hierarquia de memória.

Para que os dados possam ser processados pela C PU É NECESSÁRIO QUE ESTES ESTEJAM NOS registradores e para tal é preciso que estes dados passem por todas as camadas da hierarquia de memória até chegarem aos registradores [Zhang et al. 2015]. Como pode ser visto na Figura 1, o disco é a camada mais dista NTE E MAIS LENTA, PORÉM COM A MAIOR CAPACIDADE DE armazenamento. Com o aumento da capacidade da camada de memória principal (composta pela memória RAM) os IMDB busc AM TRAZER OS DADOS PARA ESTA CAMADA E EVITAR O NÍVEL MAIS lento da hierarquia de memória.

De NTRE A MIRÍADE DE BANCOS DE DADOS COM ARMAZENAMENTO EM MEMÓRIA, OS BANCOS chave-valor podem ser considerados os representantes mais versáteis, simples e com melhor desempenho, advindo principalmente da sua simplicidade [Pokorny 2013]. Por tanto, muitos sis TEMAS DESTA CATEGORIA SACRIFICAM GARANTIAS DE CONSISTÊNCIA EM FAVOR DO DESEMPENHO [DeCandia et al. 2007] [Fowler 2015]. N ESTES BANCOS, CADA VALOR ARMAZENADO ESTÁ VINCULADO A uma chave que identifica unicamente um valor [Han et al. 2011], sendo que este val OR PODE SER

tanto um conteúdo binário quando uma estrutura de dados lexa, conforme as funcionalidades oferecidas pelo banco [Pokorny 2013]. Nas próximas seções são apresentados e discutidos os sistemas de armazenamento chave-valor em memória Redis, Memcach**ED, VOLDEMORT,** Aerospike, Hazelcast e Riak KV.

3.1 Redis

Redis (REmote DIctionary Server) [Redis 2009] é um sistema de armazenamento de da dos estruturados em memória que pode ser utilizado como banco de dados, cache e mes sage broker [Cao et al. 2016]. Ele o**PERA EM UM MODELO CLIENTE-SERVIDOR ATRAVÉS DE CONEXÕES TCP UTILIZANDO** um protocolo próprio chamado RESP (REdis Serialization Protocol).

O modelo de dados do Redis é composto por cinco estruturas de dados diferentes para os valores (***STRING, LIST, SET, SORTED SETE HASH*), A PERSISTÊNCIA DOS DADOS DA MEMÓRIA EM DISCO** através de dois métodos (***SNAPSHOPTS*** **CHAMADOS RDB E APPEND-ONLY FILE OU AOF) [ZHANG ET AL.** 2015]. A possibilidade de escalabilidade horizontal através do Redis Cluster foi adicionada apenas em 201**5, NA VERSÃO 3.0 DO SISTEMA. SEGUNDO OS AUTORES DE [SANFILIPPO 2010], UM SISTEMA** deve ser eficiente em um único nó quando for escalado**.**

3.2 Memcached

O Memcached [Memcached 2003] caracteriza-se como um sistema genérico de cache em memória. Ele foi construído **PENSANDO NA MELHORIA DE DESEMPENHO DE APLICAÇÕES WEB ATRAVÉS** da redução na demanda de requisições ao banco de dados em disco. Brad Fitzpatrick desenvolveu ele para melhorar o desempenho do site Livejournal.com através de uma solução melhor de cache [Galbra**ITH 2009]. A SUA IMPLEMENTAÇÃO É NA LINGUAGEM PERL E POSTERIORMENTE** reescrito em C. O Memcached utiliza uma arquitetura multi thread e o controle de concorrência interno é feito através de uma hash-table estática de locks [Zhang et al. 2015]**.**

A classifica**ÇÃO DO MEMCACHED COMO BANCO DE DADOS É DISCUTÍVEL, UMA VEZ QUE O MESMO** não implementa persistência, failover [Galbraith 2009] nem escalabilidade ho rizontal, pois a distribuição dos dados entre múltiplas instâncias do sistema deve ser feita pelo cliente [Z**HANG** et al. 2015]. O funcionamento do Memcached segue o modelo cliente-servidor e a comunicação ocorre através de conexões TCP ou UDP utilizando um protocolo próprio que suporta textos puros em ASCII ou dados binários [Soliman 2013].

3.3 Voldemort

O Voldemort **[VOLDEMORT 2009] FOI DESENVOLVIDO PELO LINKEDIN EM LINGUAGEM JAVA COMO** objetivo de gerenciar funcionalidades dependentes de associações entre dados da rede social, tais como a recomendação de relacionamentos através da análise dos relaciona mentos atuais [Sumbaly et al. 2012]**.**

O Voldemort é inspirado no Dynamo, da Amazon [DeCandia et al. 2007], oferece comandos simples (***PUT, GET E DELETE*) [SUMBALY ET AL. 2013] E UMA ARQUITETURA COMPLETAMENTE** distribuída, onde cada nó é independente e não existe um servid**OR PRINCIPAL DE COORDENAÇÃO** [Deka 2014]. O sistema é completamente modularizado e tanto a seriali zação dos dados quanto a persistência são oferecidas através de módulos plugáveis. Se gundo [Sumbaly et al. 2012], a grande vantagem do Voldemort em relação a **O DYNAMO, É UM MECANISMO PRÓPRIO DE** armazenamento desenhado para o pré-carregamento de grandes volumes de dados, em que o Voldemort passa a funcionar em modo somente leitura.

3.4 Aerospike

O Aerospike [Aerospike 2012] tem uma arquitetura modelada com foco em **VELOCIDADE NA** análise de dados, escalabilidade e confiabilidade para aplicações web. Esse banco dedados se apresenta como uma solução para a combinação de diferentes tipos de dados e também acessos por milhares de usuários. Pensando nisso, as suas operaçõe**S SÃO FOCADAS EM CHAVE-VALOR E** otimizadas para o uso da memória RAM em conjunto com memórias flash (NVM) [Aerospike 2012].

Diferente de vários de seus concorrentes, o próprio Aerospike disponibiliza bibliotecas de integração aos clientes, a fim de melhorar **O DESEMPENHO NA SUA UTILIZAÇÃO. QUANTO AO CLUSTER,** todos os nós são iguais, em uma arquitetura conhecida como shared nothing. A respeito do server é possível utilizar índices secundários e definir funções para otimizar a utilização dos dados. E por fim, a **CAMADA DE ARMAZENAMENTO INCORPORA A UTILIZAÇÃO DA MEMÓRIA RAM E DE** sistemas de armazenamento permanente.

3.5 Hazelcast

O Hazelcast [Hazelcast 2009] é uma ferramenta distribuída sob licença open source e comercial desenvolvida em Java. Possui seu foco em comp**UTAÇÃO DISTRIBUÍDA E ESCALABILIDADE HORIZONTAL,** se destacando dos concorrentes por oferecer as garantias ACID (Atomi cidade, Consistência, Isolamento e Durabilidade) dos bancos de dados relacionais tradicionais.

O cluster funciona em uma arquitetura ***SHARED NOTHING*, ONDE NÃO EXISTE UM PONTO ÚNICO** de falha. Além de oferecer clientes para as linguagens comuns como Java, C, C++ e C#, o Hazelcast oferece uma API REST, está preparado para trabalhar com o protocolo de comunicação do Memcache e pode ser utilizado **ATRAVÉS DO HIBERNATE [HAZELCAST 2009].**

3.6 Riak KV

O Riak KV [Basho 2009] possui como principal objetivo oferecer disponibilidade máxima, com escalabilidade horizontal em forma de cluster, sendo considerado um banco de dados de simples operação e fácil escala**BILIDADE. EM SUA VERSÃO COMERCIAL HÁ SUPORTE A *MULTI-CLUSTER REPLICATION*,** ou seja, é possível realizar a replicação de dados através de diferentes clusters, geograficamente distantes, a fim de reduzir a latência de acesso uniformemente para clientes atravé**S DO GLOBO.**

Nota-se claramente a influência do Dynamo [DeCandia et al. 2007] no Riak KV, desde suas funcionalidades para execução distribuída até nas configurações do fator de replicação e arquitetura ***SHARED NOTHING*.**

4. Estudo comparativo

Esta seção apresen**TA UMA COMPARAÇÃO ENTRE OS SISTEMAS APRESENTADOS ANTERIORMENTE. PARA ISSO,** as características foram classificadas em três grupos**: (I) CARACTERÍSTICAS MERCADOLÓGICAS, ONDE** são explorados aspectos sem relação direta com funcionalidades ou o funci**ONAMENTO DO BANCO,** tais como ano de lançamento, licenci**AMENTO E LINGUAGEM DE DESENVOLVIMENTO; (II)** características do projeto, onde são descritas definições decididas no projeto do sistema, tais como escalabilidade, disponibilidade e consistência; e (III) característic**AS DE MANUTENÇÃO, ONDE** são explanadas os aspectos que te**M RELAÇÃO DIRETA COM A MANUTENÇÃO E SUPORTE AO SISTEMA, TAIS** como ferramentas internas para monitoramento e interface de gerenciamento.

Na Tabela 1 é possível avaliar: o ano em que a primeira versão d**O SISTEMA FOI** lançada, os licenciamentos sob os quais o software é distribuído, a linguagem na qual o sistema foi desenvolvido, os sistemas operacionais suportados, as linguagens nas quais são oferecidos clientes para comunicação e os protocolos de comunic**AÇÃO SUPORTADOS. A PARTIR DOS DADOS** disponibilizados, os interessados podem avaliar a maturidade do sistema, se a licença está alinhada com as necessidades empresariais e se a infraestrutura disponível e o esforço de implementação estão dentro do esperado.

Vale ressaltar que o Redis não suporta oficialmente Windows, mas uma versão para Windows x64 é mantida pela equipe da MS Open Tech (Microsoft Open Technolo gies). Quanto ao Voldemort e ao Hazelcast, como os mesmos rodam na JVM (Java Virtual Machine), pode**M-SE CONSIDERAR OS SISTEMAS OPERACIONAIS SUPORTADOS POR ESTA TECNOLOGIA TANTO** Aerospike quanto Riak KV oferecem pacotes para sistemas baseados nas distribui- ções Linux Red Hat, Debian e Ubuntu. O Aerospike ainda oferece sua execução no OS X e Windows atr**AVÉS DE MÁQUINAS VIRTUAIS.**

Tabela 1. Características mercadológicas

	Redis	Memcached	Voldemort	Aerospike	Hazelcast	Riak KV
Lançamento	2009	2003	2009	2012	2009	2009
Licenciamento	BSD-3 e comercial	BSD-3	Apache 2	AGPL e co**MERCI** al	Apache 2 e comercial	Apache 2 e comercial
Desenvolvido	C	C	Java	C	Java	Erlang
SO Suporte	Linux, BSD, OSX e Windows	Debian/Ubuntu e **WINDOWS**	JVM	Linux, OS X e Windows	JVM	Linux
Clientes	48 linguagens	Não existe listagem oficial	4 linguagens	12 lingua**GEN** s	6 linguagens	21 linguagens
Protocolos	Próprio (RESP)	Próprio	HTTP, Socket, NIO	Próprio e JDBC	Próprio e Memcached	API HTTP e próprio

Quanto ao item linguagens com cliente, é importante notar que foram considera dos apenas as linguagens e cl**IENTES LISTADOS NO SITE OFICIAL DE CADA SISTEMA E QUE O SITE DO** Memcached não oferece uma listagem oficial das lin**GUAGENS SUPORTADAS. NOTA-SE TAMBÉM QUE** as linguagens C++, Java e Python são as únicas para as quais todos os sistemas possuem clientes. Os pro**TOCOLOS DE COMUNICAÇÃO SÃO O MEIO DE COMUNICAÇÃO ENTRE O CLIENTE E O** servidor, porém, na maioria dos casos a comunicação é feita através de um dos clientes já construídos e a empresa não precisa se preocupar com o protocolo utilizado pelo cliente

Na Tabela **2 É POSSÍVEL AVALIAR: AS OPÇÕES PARA ESCALABILIDADE HORIZONTAL (OU** clusterização), a classificação do sistema segundo o teorema CAP [Brewer 2000], como é feito o controle de concorrência, o suporte à transações ACID, as opções de persistência dos dados em**DISCO, O SUPORTE A DADOS COMPLEXOS E AS OPÇÕES PARA AUTENTICAÇÃO DO CLIENTE. ANALISANDO** a Tabela 2 é possível avaliar se as funcionalidades e características do banco de dados atendem às demandas e características referentes aos dados que se pretende arma**ZENAR NOS MESMOS.**

Quanto à escalabilidade, todos os bancos exceto o Memcached incluem suporte a *sharding* **E REPLICAÇÃO, SENDO QUE A MAIORIA (A EXEMPLO DO DYNAMO [DECANDIA ET AL. 2007])** oferecem fator de replicação configurável para leituras e escritas. O Re**DIS UTILIZA O MODELO** *master/slave*, **COMUMENTE UTILIZADO NAS BASES DE DADOS RELACIONAIS TRADICIONAIS.**

O teorema CAP (***CONSISTENCY, AVAILABILITY E PARTITION TOLERANCE***) foi proposto por Eric Brewer em [Brewer 2000] e verificado em [Gilbert and Lynch 2002], desd**E ENTÃO PASSOU A SER LARGAMENTE ACEITO PELA ACADEMIA. O** teorema afirma que na existência de uma falha de comunicação (***PARTITION***) **CADA NÓ DE UM** sistema distribuído deve escolher entre responder requisições, mantendo a disponibilidade (***AVAILABILITY***) **E ASSUMINDO O RISCO DE NÃO RETORNAR OS DADOS MAIS ATUAIS, OU REJEITAR** requisições para garantir a consistência dos dados (***CONSISTENCY***)**. SISTEMAS CLASSIFICADOS COMO** AP priorizam a disponibilidade, enquanto que sistemas classificados como CP priorizam a consistência**. O TEOREMA TEM SIDO ALVO DE MUITAS CRÍTICAS E BREWER EXPLORA ALGUMAS DE SUAS** limitações em [Brewer 2012], enquanto Abadi propõe o teorema PACELC como alternativa em [Abadi 2012]**.**

Quanto à classificação do Redis, é importante mencionar que o ele não atende **TODOS OS** requisitos de um sistema CP de [Brewer 2000], por usar replicação assíncrona entre os nós do cluster. O teorema CAP também não se aplica ao Memcached pelo fato de ele não suportar a criação de clusters e, portanto, não haver comunicação entre nós**. O RIAK KV POSSUI UMA** configuração onde é possível definir qual dos atributos devem ser preservados (disponibilidade ou consistência).

Tabela 2. Características do projeto

	Redis	Memcached	Voldemort	Aerospike	Hazelcast	Riak KV
Escalabilidade horiont**AL**	Mestre - Escravo	Não	Fator de Replicação	Fator de Replicação	Fator de Replicação	Fator de Replicação
Teorema CAP	CP	N/A	AP	AP	AP	Config.
Controle Concorrência	Single thread	Mutex lock	MVCC	Test-and-set	Multi-single-thread	MVCC
Transações	Parcial	Não	Não	Parcial	Sim	Não
Persistência em d**ISCO**	RDB e AOF	Não	Config.	Assínc.	Banco auxi-liar	Banco auxi**LIAR**
Suporte a dados complexos	Sim	Não	Sim	Sim	Sim	Sim
Autenticação	Simples	SASL	Kerberos	Somente comercial	Simples, SS**L,** Kerberos,IP	Sim, e autorização

O controle de concorrência é implementado de diferentes maneiras. No Redis a execução é single-thread e as requisições são processadas de forma assíncrona in ternamente [Zhang et al. 2015], sendo que apenas um ***MASTER*** **RESPONDE POR UMA DETER MINADA CHAVE,** portanto não há concorrência. O Memcached utiliza ***MUTEX*** **(*MUTUAL EXCLUSIVE*)** ***LOCK*****. TANTO** Voldemort quanto Riak KV seguem a implementação do Dynamo [DeCandia et al. 2007] e utilizam ***VECTOR CLOCKS*****, UMA IMPLEMENTAÇÃO DO VERSIONAMENTO BASEADO EM *LOCKING* OTIMISTA** conhecida por MVCC (Multi Version Concurrency Control). O Aerospike utiliza o método conhecido como ***TEST-AND-SET*** **OU *CHECK-AND-SET* (CAS), UMA OPERAÇÃO ATÔMICA IMPLEMENTADA A** baixo nível que escreve em um local de memória e **RETORNA O VALOR ANTIGO. NO HAZELCAST, É CRIADA** uma ***THREAD*** **PARA ATENDER CADA UMA DAS PARTIÇÕES INTERNAS DE DADOS, PORTANTO AINDA QUE ELE** seja ***MULTI-THREAD*****, UMA CHAVE ESPECÍFICA ESTÁ NUM CONTEXTO SINGLE-THREAD E, PORTANTO, NÃO HÁ** concorrência

Quanto as tran**SAÇÕES, HÁ UM NÍVEL BASTANTE VARIADO DE SUPORTE OFERECIDO PELOS SISTEMAS.** Ainda que o Redis tenha suporte básico a transações, estas não possuem opção de ***ROLLBACK*** **E A** durabilidade da mesma depende da persistência em disco. Memcached, Voldemort e Riak KV de**CLARAM NÃO SUPORTAREM TRANSAÇÕES, ENQUANTO QUE O AEROSPIKE SUPORTA TRANSAÇÕES QUE** envolvam uma única chave ou que sejam somente leitura, no caso de envolverem múltiplas chaves. O Hazelcast se destaca sendo o único a oferecer transações ACID completas**.**

A persistência em disco é oferecida por todos os bancos, exceto o memcached. No Redis são oferecidas duas formas complementares: snapshot (RDB), onde todos os dados na memória são gravados em disco, e append-only file (AOF), onde cada operação é gravada em um arquivo de log e o arquivo é reescrito quando chega em um tamanho pré-determinado. O Voldemort oferece opções para configurar a persistência como síncrona (*write through*, onde a operação é persistida antes do retorno ao cliente) ou assíncrona (*write behind*, onde o cliente recebe a confirmação e posteriormente a operação é persistida), enquanto que o Aerospike faz a persistência de forma assíncrona. Vale mencionar que o Aerospike tem melhorias focadas no uso de SSD como dispositivo de armazenamento permanente. Tanto Hazelcast como Riak KV oferecem persistência através do acoplamento de um banco de dados auxiliar e a sincronicidade é configurável.

Referente ao suporte a dados complexos, vale notar que todos os sistemas, exceto o Memcached, suportam chaves do tipo lista e hashtable (ainda que com nomes diferentes). Hazelcast e Voldemort se baseiam fortemente nas classes do Java, Redis e Riak KV ainda oferecem suporte ao tipo HyperLogLogs, enquanto Redis e Aerospike oferecem suporte a tipagem ou comandos relativos a georreferenciamento.

Finalizando, enquanto que Redis oferece autenticação simples através de credenciais pré-configuradas, o Memcached oferece autenticação através do protocolo SASL(*Simple Authentication and Security Layer*). O Voldemort é integrado ao protocolo Kerberos. O Aerospike oferece autenticação apenas em sua versão com licenciamento comercial. O Hazelcast oferece todos os protocolos supracitados e o SSL. Por último, o Riak KV oferece um sistema próprio de usuários e grupos, com autenticação e autorização baseada em diversos mecanismos, incluindo senhas e certificados digitais.

Na Tabela 3 está descrita a existência de interfaces de gerenciamento, ferramentas de monitoramento e benchmarks para os sistemas estudados. É importante notar que fo ram avaliadas apenas as ferramentas oficiais dos desenvolvedores dos sistemas. Portanto, muitas destas ferramentas, ainda que não estejam declaradas aqui, já foram desenvolvidas pela comunidade e estão disponíveis. Estas informações são particularmente interessantes para avaliar o esforço de manutenção que será despendido após a implantação do sistema.

Tabela 3. Características de manutenção

	Redis	Memcached	Voldemort	Aerospike	Hazelcast	Riak KV
Interface de Gerenciamento	Não	Não	Básica	À parte	Comercial	Sim
Ferramentas de Monitoramento	'INFO'	'stats'	JMX	'asadm'	JMX	'stats'
Benchmark embutido	Sim	Não	Sim	Sim	Não	Sim

Quanto às interfaces de gerenciamento oferecidas pelos sistemas avaliados, o Redis e o Memcached são os únicos que não as oferecem nativamente (ainda que existam opções na comunidade) e o Voldemort oferece uma interface básica à parte escrita em Ruby, que está sem manutenção. O Aerospike oferece uma interface que deve ser instalada à parte. No Hazelcast, esta funcionalidade está disponível apenas na versão comercial. O Riak KV é o único onde a interface de gerenciamento já está integrada ao código principal do programa, não requerendo nenhuma instalação extra.

A respeito de ferramentas de monitoramento, o Redis oferece comandos como INFO, MEMORY e LATENCY, enquanto que o Memcached oferece o comando stats e o Aerospike oferece o comando asadm. O Voldemort possui uma interface completa de monitoramento exposta através de Java Management Extensions (JMX). Esta é a mesma estratégia utilizada pelo Hazelcast. O Riak KV oferece os comandos status e stats em sua interface de linha de comando (CLI) riak-admin e a URL /stats em sua API HTTP.

No item benchmark embutido foi avaliado se são disponibilizados benchmarks junto com o sistema, cujo principal objetivo é **AVALIAR O DESEMPENHO DO SISTEMA EM DETER MINADA** infraestrutura. Enquanto que Memcached e Hazelcast não oferecem ferramentas próprias para realização de benchmark, no Redis existe a ferramenta redis-benchmark e o Voldemort oferece a ***VOLDEMORT-PERFORMANCE-TOOL*. NO AEROSPIKE OS *BENCHMARKS* ESTÃO NOS CLIENTES** disponibilizados e no Riak KV o nome dado ao benchmark é Basho Bench**.**

Após comparar as características dos bancos de dados chave-valor com armazenamento em memória, Redis, Memcached, Voldemort, Aerospike, **HAZELCAST E RIAK KV, É FÁCIL ENTENDER** o motivo pelo qual o Memcached não é considerado um banco de dados, pois seu foco destoa bastante de seus semelhantes. É possível perceber também que, mesmo que o Redis tenha oferecido suporte à clusterização em suas v**ERSÕES MAIS RECENTES, OS OUTROS SISTEMAS AINDA ESTÃO** à frente quando o assunto é funcionalidades para execu- ção distribuída. É possível perceber também como Voldemort e Hazelcast se utilizam do ecossistema Java para prover funcionalidades interessantes e **COMO O *PAPER* DO DYNAMO [DECANDIA ET AL. 2007] INFLUENCIA** principalmente Voldemort e Riak KV.

5. Conclusões

Após estudar os bancos chave-valor com armazenamento em memória, é possível notar que mesmo um subconjunto específico de bancos NoSQL traz muitas variáv**EIS. NESTE SENTIDO,** destaca-se a grande quantidade de características que devem ser cuidadosamente avaliadas pelo analista para a correta tomada de decisão quanto ao banco mais adequado às necessidades. Dentre estas características, destacam-se o padrão de **BUSCA E GRAVAÇÃO DE DADOS DA APLICAÇÃO** cliente, a importância da durabilidade dos dados, o comportamento desejado frente a partições no cluster, o ambiente de infraestrutura onde a solução será implantada, o ambiente de desenvolvimento da aplicação client**E E AS PERSPECTIVAS DE CRESCIMENTO NA DEMANDA DA APLICAÇÃO.**

Com as características dos sistemas esclarecidas, percebe-se que outro fator importante para a escolha do banco de dados chave-valor com armazenamento em memória a ser adotado é o desempenho, que **É JUSTAMENTE O PONTO QUE TRAZ MAIS INTERESSE A ESTA CATEGORIA DE SISTEMAS.** Como trabalho futuro, propõe-se uma avaliação do desempenho dos sistemas aqui estudados, comparando o desempenho das **VARIADAS CARACTERÍSTICAS COMPARTILHADAS PELOS MESMOS.**

6. Referência**S**

[Abadi 2012] Abadi, D. (2012). Consistency Tradeoffs in Modern Distributed Database System Design: CAP is Only Part of the Story. Computer, 45(2):37–42**.**

[Aerospike 2012] Aerospike (2012). Aerospike | High Performance NoSQL Database. Access on <http://www**.AEROSPIKE.COM/>.**

[Basho 2009] Basho (2009). Riak KV. Access on <http://basho.com/products/riak-kv/>

[Brewer 2000] Brewer, E. (2000). Towards Robust Distributed Systems. In Proceedings **OF THE** Nineteenth Annual ACM Symposium on Principles of Distributed Com**PUTING, PODC '00,** pages 7–, New York, NY, USA. ACM.

[Brewer 2012] Brewer, E. (2012). CAP twelve years later: How the "rules"have changed. Computer, 45(2):23–29

[Cao et al. 2016] Cao, W., Sahin, S., Liu, L., and Bao, X. (2016). Evaluation and Analysis of In-**MEMORY KEY-VALUE SYSTEMS. IN 2016 IEEE INTERNATIONAL CONGRESS ON BIG DATA (BIG DATA** Congress), pages 26–33.

[Carlson 2013] Carlson, J. L. (2013). Redis in Action. Manning, Shelter Island, NY, USA**.**

[Chang et al. 2008] Chang, F., Dean, J., Ghemawat, S., Hsie**H, W. C., WALLACH, D. A.,** Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E. (2008). Bigtable: A distributed storage system for structured data. ACM Trans. on Computer Systems (TOCS), 26(2):4.

[DeCandia et al. 2007] DeCandia, G., Hastorun, D., Jampani, **M., KAKULAPATI, G., LAKSHMAN,** A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. (2007). Dynamo: amazon's highly available key-value store. ACM SIGOPS operating systems review, 41(6):205–220.

[Deka 2014] Deka, G. C. (2014). A survey of clo**UD DATABASE SYSTEMS. IT PROFESSIONAL,** 16(2):50–57.

[Fowler 2015] Fowler, A. (2015). NoSQL For Dummies. John Wiley & Sons, 111 River Street, Hoboken, New Jersey, USA.

[Galbraith 2009] Galbraith, P. (2009). Developing Web Applications with Apache, MySQL, mem**CACHED, AND PERL. JOHN WILEY & SONS.**

[Gilbert and Lynch 2002] Gilbert, S. and Lynch, N. (2002). Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services. SIGACT News, 33(2):51–59.

[Han et al. 2011] Han, J., E, H., L**E, G., AND DU, J. (2011). SURVEY ON NOSQL DATABASE. IN 2011** 6th International Conference on Pervasive Computing and Applications, pages 363–366.

[Hazelcast 2009] Hazelcast (2009). Hazelcast the Leading In-Memory Data Grid - Hazelcast.com. Access on <https:**//HAZELCAST.COM/>.**

[Hecht and Jablonski 2011] Hecht, R. and Jablonski, S. (2011). NoSQL evaluation: A use case oriented survey. In 2011 International Conference on Cloud and Service Computing (CSC), pages 336–341.

[Kasavajhala 2011] Kasavajhala, V. (2011). **SOLID STATE DRIVE VS. HARD DISK DRIVE PRICE AND** Performance Study. Proc. Dell Technical White Paper, pages 8–9.

[Memcached 2003] Memcached (2003). memcached - a distributed memory object caching system. Access on <https://memcached.org/>.

[Pokorny 2013] P**OKORNY, J. (2013). NOSQL DATABASES: A STEP TO DATABASE SCALABILITY IN WEB** environment. International Journal of Web Information Systems, 9(1):69–82.

[Redis 2009] Redis (2009). Redis.io. Access on <http://redis.io/>.

[Robbins 2008] Robbins, S. (2008). RAM i**S THE NEW DISK... ACCESS ON** <https://www.infoq.com/news/2008/06/ram-is-disk>.

[Sanfilippo 2010] Sanfilippo, S. (2010). On Redis, Memcached, Speed, Benchmarks and The Toilet . Access on <http://antirez.com/post/redis-memcached-benchmark.html>.

[Schroeder et **AL. 2016] SCHROEDER, B., LAGISETTY, R., AND MERCHANT, A. (2016). FLASH RELIABILITY** in Production: The Expected and the Unexpected. In Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST '16), FAST '16, pages 67–80, Santa Clara**, CA, USA.**

[Soliman 2013] Soliman, A. (2013). Getting Started with Memcached. Packt.

[Sumbaly et al. 2012] Sumbaly, R., Kreps, J., Gao, L., Feinberg, A., Soman, C., and Shah, S. (2012). Serving large-scale batch computed data with Project Voldemort. In Pro**CEEDINGS OF THE** 10th USENIX conference on File and Storage Technologies, page 18.

[Sumbaly et al. 2013] Sumbaly, R., Kreps, J., and Shah, S. (2013). The Big Data Ecosystemat LinkedIn. In Proceedings of the 2013 ACM SIGMOD International Conference on Manage**MENT OF DATA, SIGMOD '13, PAGES 1125–1134, NEW YORK, NY, USA. ACM.**

[Voldemort 2009] Voldemort (2009). Project Voldemort. Access on <http://www.projectvoldemort.com/>.

[Zhang et al. 2015] Zhang, H., Chen, G., Ooi, B. C., Tan, K.-L., and Zhang, M. (2015). In-Memory Big Data Management and Processing: A Survey. IEEE Transactions on Knowledge and Data Engineering, 27(7):1920–1948.

Apêndice **D** ou online no GitHub, em https://github.com/dineiar/monitor_cpp/.

3.5.3 Instalação e configuração dos bancos de dados chave-valor

Cada banco de dados possui seu método de instalação, de acordo com o sistema operacional utilizado. Na sequência estão apresentados os passos seguidos para instalação dos bancos de dados chave-valor Redis e Aerospike no ambiente de testes.

3.5.3.1 Redis

Para instalar o Redis no Ubuntu foi seguida a recomendação oficial de compilar o código-fonte e instalá-lo como um serviço para inicializar automaticamente. Na Figura 5 é possível observar os comandos executados para efetuar a instalação deste banco de dados.

Figura 5 - Comandos para instalação do Redis

```
> sudo apt-get update
> sudo apt-get install build-essential tcl
> cp /tmp
> wget http://download.redis.io/redis-stable.tar.gz
> tar xzvf redis-stable.tar.gz
> cd /tmp/redis-stable
> make
> make test
> sudo make install
> sudo mkdir /etc/redis
> sudo mkdir /var/redis
> sudo mkdir /var/redis/6379
> sudo cp /tmp/redis-stable/redis.conf /etc/redis/6379.conf
```

Após executar os passos da Figura 5, foi necessário preparar o Redis para aceitar comunicações externas e executá-lo de forma independente, para isso é preciso abrir o arquivo de configuração */etc/redis/6379.conf* e efetuar as seguintes alterações:

- Comentar a linha "*bind 127.0.0.1*", colocando um caractere de # no início da mesma.
- Habilitar a execução em modo daemon ("*daemonize yes")*
- Alterar o supervisor do processo ("*supervised systemd*")
- Alterar o *working directory* ("*dir /var/redis/6379*")
- Desabilitar o modo protegido ("*protected-mode no*")

Após estas alterações, para que seja possível iniciar o Redis como um serviço é preciso ainda executar os comandos apresentados na Figura 6.

Figura 6 - Comandos para configurar o Redis como serviço

```
> sudo cp /tmp/redis-stable/utils/redis_init_script /etc/init.d/redis_6379
> sudo update-rc.d redis_6379 defaults
> sudo service redis_6379 start
```

Por fim, basta realizar um teste executando o comando "*redis-cli ping*", sendo que o retorno deve ser "*pong*". Além destas configurações de instalação, para execução dos testes ainda foram feitas algumas alterações a respeito da persistência que estão descritas no item 3.5.3.3.

3.5.3.2 Aerospike

Para instalar o Aerospike, seguiu-se a lista de comandos apesentados na Figura 7.

Figura 7 - Comandos para instalação do Aerospike

```
> wget -O aerospike.tgz
'http://www.aerospike.com/download/server/3.12.1.1/artifact/ubuntu16'
> tar -xvf aerospike.tgz
> cd aerospike-server-community-3.12.1.1-ubuntu16.04/
> sudo ./asinstall
```

Após isso é necessário abrir o arquivo "/etc/aerospike/aerospike.conf", e adicionar o *namespace ycsb*, conforme Figura 8.

Figura 8 - Adição de um namespace no Aerospike

```
namespace ycsb {
      memory-size 16G
      replication-factor 1
      high-water-memory-pct 95
      stop-writes-pct 98
      default-ttl 0
}
```

3.5.3.3 Configurações de persistência e durabilidade

Foram identificadas quatro formas de persistência em que os bancos poderiam ser comparados de forma igualitária:

1. Não utilizar persistência, trabalhando somente em memória;
2. Ativar o log *append-only*, deixando o *fsync* a cargo do sistema operacional (configuração padrão do Riak KV);
3. Ativar o log *append-only*, fazendo *fsync* a cada segundo (configuração padrão do Redis);
4. Ativar o log *append-only*, fazendo *fsync* a cada escrita.

Dentre estas formas, os autores escolheram realizar os testes com a primeira opção, sem persistência, por entenderem que este é um caso de uso bastante comum.

Nesta forma, para a limpeza dos bancos entre cada teste bastou reiniciar o serviço do mesmo, pelo fato de os dados estarem apenas na memória.

3.5.4 Instalação e configuração dos bancos de dados orientados a documentos

Na sequência estão apresentados os passos seguidos para instalação dos bancos de dados orientados a documentos MongoDB e Couchbase no ambiente de testes.

3.5.4.1 MongoDB

Para instalar o MongoDB foi seguida a documentação oficial, e os comandos executados estão resumidos na Figura 9.

Figura 9 - Comandos para instalação do MongoDB

```
> sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv
0C49F3730359A14518585931BC711F9BA15703C6
> echo "deb [ arch=amd64,arm64 ] http://repo.mongodb.org/apt/ubuntu
xenial/mongodb-org/3.4 multiverse" | sudo tee
/etc/apt/sources.list.d/mongodb-org-3.4.list
> sudo apt-get update
> sudo apt-get install -y mongodb-org
> sudo service mongod start
```

Feito isto, basta verificar o final do arquivo “/var/log/mongodb/mongod.log” para verificar se o banco está rodando normalmente.

O benchmark YCSB possui integração com os *drivers* síncrono e assíncrono do MongoDB. Nos testes realizados neste trabalho foi utilizado o *driver* síncrono, de forma a manter o padrão com os outros bancos.

3.5.4.2 Couchbase

Os comandos executados para instalar o Couchbase e criar o *bucket* “*ycsb*” utilizado nos testes estão documentados na Figura 10. O IP “11.11.11.110” refere-se ao IP do servidor onde o Couchbase está rodando.

Figura 10 - Comandos para instalação do Couchbase

```
> wget https://packages.couchbase.com/releases/4.5.1/couchbase-server-
community_4.5.1-ubuntu14.04_amd64.deb
> dpkg -i couchbase-server-community_4.5.1-ubuntu14.04_amd64.deb
> /opt/couchbase/bin/couchbase-cli bucket-create -c 11.11.11.110:8091 -u
admin -p password --bucket=ycsb --enable-index-replica=0 --bucket-
type=couchbase --bucket-priority=low --bucket-eviction-policy=valueOnly --
bucket-ramsize=6446 --bucket-replica=0
```

Pode-se acessar a interface de gerenciamento e monitoramento na porta 8091 para verificar o funcionamento do banco e a situação do *bucket* criado.

3.5.4.3 Configurações de persistência e durabilidade

Os bancos de dados orientados a documentos MongoDB e Couchbase suportam diferentes formas de persistência e durabilidade.

Ainda que o Couchbase não utilize a técnica de *journaling* e sua configuração padrão seja executar o comando em memória e gravar os dados em disco de forma assíncrona (o retorno ao cliente não aguarda a persistência acontecer), é possível configurá-lo para persistir os dados de forma síncrona através da opção "*PersistTo*".

O MongoDB utiliza-se da técnica de *journaling*, porém sua configuração padrão é fazer esta escrita ao *journal* também de forma assíncrona, não aguardando a confirmação da escrita antes do retorno ao cliente (configurável através da opção "*j*" na *string* de conexão). O padrão também é fazer o *fsync* (sincronização com o disco) do *journal* a cada 50 milissegundos, sendo possível alterar este tempo através da configuração "*storage.journal.commitIntervalMs*".

Nota-se que nenhum dos bancos atende aos requisitos de durabilidade no conceito ACID, pois em ambos os casos é possível que a operação seja perdida após o retorno com sucesso ao cliente, ainda que seja um caso raro.

Ambos os bancos foram testados com suas configurações padrão, haja visto que são semelhantes nas garantias que oferecem e o desempenho do disco não afeta diretamente o *throughput*.

3.5.5 Instalação e configuração do Cassandra

Ainda que seja desenvolvido em Java, assim como outros bancos avaliados neste estudo, o Cassandra possui algumas limitações no sentido de que deve ser utilizado o JDK da Oracle e não o OpenJDK. Para baixar o JDK da Oracle é preciso criar uma conta no site da mesma e fazer o download aceitando os termos especificados. Os passos documentados na Figura 11 descreve os passos para a instalação do JDK da Oracle, e assume que o pacote foi baixado do site oficial e colocado na pasta atual.

Figura 11 - Comandos para instalação do JDK da Oracle

```
> sudo mkdir -p /usr/lib/jvm
> sudo tar zxvf jdk-8u131-linux-x64.tar.gz -C /usr/lib/jvm
> sudo update-alternatives --install "/usr/bin/java" "java"
"/usr/lib/jvm/jdk1.8.0_131/bin/java" 1
> sudo update-alternatives --set java /usr/lib/jvm/jdk1.8.0_131/bin/java
```

Tendo o Oracle JDK instalado pode-se prosseguir com a instalação do Cassandra através dos comandos documentados na Figura 12

Figura 12 - Comandos para instalação do Cassandra

```
> echo "deb http://www.apache.org/dist/cassandra/debian 22x main" | sudo
tee -a /etc/apt/sources.list.d/cassandra.sources.list
> curl https://www.apache.org/dist/cassandra/KEYS | sudo apt-key add -
> sudo apt-get update
> sudo apt-get install cassandra
```

A configuração do cluster foi feita através da edição do arquivo *cassandra-topology.properties*, apontando o servidor como membro único de um cluster através da linha "*192.168.1.1=DC1:RAC1*" onde 192.168.1.1 é o IP do servidor. Para configurar o banco de dados, por sua vez, foi editado o arquivo *cassandra.yaml*, definindo *listen_address: 192.168.1.1* (para permitir conexões externas), *auto_snapshot: false* (para desativar o backup na exclusão dos dados) e *concurrent_writes: 64* (para não limitar os testes pelo I/O de disco).

A criação do *keyspace* e da tabela foi feita através da ferramenta *cqlsh*, distribuída junto com o Cassandra. Os comandos utilizados estão descritos na Figura 13.

Figura 13 - Criação do *keyspace* e tabela no Cassandra pelo *cqlsh*

```
> CREATE KEYSPACE ycsb WITH REPLICATION = { 'class' : 'SimpleStrategy',
'replication_factor' : 1 };
> USE ycsb;
> CREATE TABLE usertable (y_id varchar primary key,field0 varchar,field1
varchar,field2 varchar,field3 varchar,field4 varchar,field5 varchar,field6
varchar, field7 varchar,field8 varchar,field9 varchar);
```

3.5.6 Execução dos testes

Este trabalho de pesquisa possui como um de seus objetivos medir o desempenho de aplicativos de bancos de dados NoSQL. Para isso foram formuladas algumas suposições referindo-se à distribuição da probabilidade de uma ou mais populações. Na sequência foram executados os testes de hipóteses rejeitando ou validando as mesmas. Esta seção irá descrever o planejamento e execução dos testes.

3.5.6.1 Configuração formal dos experimentos

A métrica de *troughput*, fornecida pelo *benchmark* é obtida através da divisão do número total de operações, pelo total do tempo de execução das operações, a fórmula abaixo representa essa medida, onde *Th* representa *Throughput, o(n)* o total de operações e *t(s)* o tempo de execução.

$$Th = o(n)/t(s)$$

A latência é fornecida pelo *benchmark*, ela representa o tempo de resposta do servidor para com o cliente para cada requisição enviada, e é calculada através da média aritmética da soma da diferença do *timestamp* final (*tf*) e *timestamp* inicial (*ti*) de cada operação de leitura e inserção. A fórmula abaixo representa a latência (*Lt*).

$$Lt = \frac{\sum(tf - ti)}{o(n)}$$

O tempo total de execução (*Rt*), é fornecido pelo *benchmark* e se refere a diferença do *timestamp* final (*tf*) da carga de dados com o *timestamp* inicial (*ti*) da carga total de dados.

$$Rt = tf - ti$$

3.5.6.2 Caracterização formal das hipóteses

Esta seção apresenta as hipóteses estatísticas que foram levantadas para realizar os testes. Em relação as variáveis das fórmulas, para bancos de dados chave-valor *a* é Redis e *b* equivalente a Aerospike. Já para bancos de dados orientados a documentos *a* é MongoDB e *b* representa Couchbase. Na comparação Cassandra e Couchbase *a* é Cassandra e *b* representa Couchbase.

Em relação a normalidade do *Throughput* as hipóteses são as seguintes:

$$H_0 - Th_a = Th_b$$

Já as hipóteses alternativas são:

$$H_1 - Th_a > Th_b$$

$$H_2 - Th_a < Th_b$$

Em relação a métrica de latência, que se trata da diferença do tempo de resposta do servidor para com o cliente, as hipóteses de normalidade são as seguintes:

$$H_0 - Lt_a = Lt_b$$

Já as hipóteses alternativas são:

$$H_1 - Lt_a > Lt_b$$

$$H_2 - Lt_a < Lt_b$$

Em relação a métrica de *run time*, que se refere ao tempo de execução das iterações. As hipóteses são:

$$H_0 - Rt_a = Rt_b$$

Já as hipóteses alternativas são:

$$H_1 - Rt_a > Rt_b$$

$$H_2 - Rt_a < Rt_b$$

3.5.6.3 Intervalo de confiança

O intervalo de confiança é responsável por definir a taxa de sucesso de um procedimento experimental. Para este experimento o intervalo de confiança utilizado foi de 95%.

3.6 RESULTADOS DOS EXPERIMENTOS

3.6.1 Análise e avaliação *do throughput*, latência e tempo de execução

Nesta seção estão apresentados e discutidos os resultados da avaliação das métricas de *Throughput*, latência e tempo de execução dos bancos de dados avaliados, separando em subseções de acordo com a classificação do banco de dados.

Além da avaliação realizada por categorias, foi considerado uma avaliação do Couchbase e do Cassandra, isso é relevante em função das possíveis aplicabilidades destes modelos para os mesmos fins, comparações similares foram realizadas por Abramova, Bernardino e Furtado (2014), assim como Kabakus e Kara (2016).

3.6.1.1 Throughput dos bancos de dados chave-valor

O *Throughput* representa o número de operações por segundo que o banco de dados foi capaz de processar durante o tempo de execução do teste. Neste caso foi coletada uma amostra de 32 iterações. Os resultados obtidos estão apresentados no Quadro 16.

Quadro 16 - *Throughput* dos bancos de dados chave-valor

Iteração	Redis	Aerospike	Iteração	Redis	Aerospike
1	18422,31	54709,38	17	19052,48	51738,41
2	20725,39	54552,40	18	19390,74	54695,02
3	18971,80	52434,54	19	19622,23	53956,06
4	20636,26	54854,64	20	19162,30	53070,67
5	20812,17	53429,65	21	20946,27	52634,35
6	19991,84	55035,17	22	21394,13	55249,84
7	20240,21	55531,49	23	20622,21	54519,68
8	21841,02	53659,01	24	19052,12	54572,05
9	20793,74	54826,36	25	20173,57	53233,96
10	19779,19	53598,04	26	20004,96	50920,13
11	20842,19	52881,51	27	20026,76	53030,70
12	21436,59	51747,51	28	20107,37	55356,89
13	19538,73	51625,69	29	20809,92	53151,91
14	21638,09	52297,97	30	20533,80	55705,96
15	20788,90	53791,78	31	20307,04	55109,17
16	20340,25	53987,52	32	20784,66	53949,65

Com estes resultados é possível obter várias informações, sobre as operações por segundo utilizando os mais diversos testes estatísticos. Sendo assim, utilizou-se o SPSS para realizar a análise estatística dos dados, a qual está apresentada no Quadro 17.

Quadro 17 –*Throughput* dos bancos chave-valor: análise descritiva gerada pelo SPSS

			Estatística	Erro Padrão
Throughput Redis	Média		20274,664598303690000	146,144749325722330
	95% Intervalo de Confiança para Média	Limite inferior	19976,600416933680000	
		Limite superior	20572,728779673700000	
	5% da média aparada		20283,333126458303000	
	Mediana		20323,647106644145000	
	Variância		683465,208	
	Desvio Padrão		826,719546264211100	
	Mínimo		18422,313105633544000	
	Máximo		21841,023557728010000	
	Intervalo		3418,710452094466700	
	Intervalo interquartil		1144,405400388212000	
	Assimetria		-,268	,414
	Curtose		-,343	,809
Throughput Aerospike	Média		53745,534816535050000	223,891046116127030
	95% Intervalo de Confiança para Média	Limite inferior	53288,906017453504000	
		Limite superior	54202,163615616600000	
	5% da média aparada		53782,524819355410000	
	Mediana		53870,718474990580000	
	Variância		1604070,417	
	Desvio Padrão		1266,519015645307700	
	Mínimo		50920,126689275200000	
	Máximo		55705,961652016000000	
	Intervalo		4785,834962740795000	
	Intervalo interquartil		1878,308160288717800	
	Assimetria		-,424	,414
	Curtose		-,650	,809

Com está análise é possível observar que a média de operações por segundo do Aerospike é maior que o dobro da média do Redis, estando 95% confiante que o valor da média em uma execução futura deste cenário de teste, com 32 ou mais iterações, ficará entre 53288,90 e 54202,16 para o Aerospike e entre 19976,60 e 20572,72 para o Redis.

De acordo com o que foi definido no experimento, é necessário avaliar a normalidade das informações, para identificar qual método de abordagem estatística deve ser aplicada no teste. Quando as abordagens Kolmogorov-Smirnov e Shapiro-Wilk, apontam Sig. (Significância) maior que 0,05, indica que existe homogeneidade e normalidade nas amostras avaliadas, sendo assim o teste deve ser paramétrico, caso o Sig. seja menor que 0,05, então o teste deve ser não paramétrico.

Conforme o Quadro 18 apresenta, estas amostras estão distribuídas normalmente e existe homogeneidade, isso pode ser concluído observando que o Sig. é maior do que 0,05. O que indica a utilização de um teste paramétrico, nesse caso o teste utilizado foi Teste-T de pares.

Quadro 18 - Teste de normalidade do *Throughput* dos bancos chave-valor

	Kolmogorov-Smirnov[a]			Shapiro-Wilk		
	Estatística	df	Sig.	Estatística	df	Sig.
Throughput Redis	,100	32	,200*	,975	32	,639
Throughput Aerospike	,136	32	,141	,962	32	,317

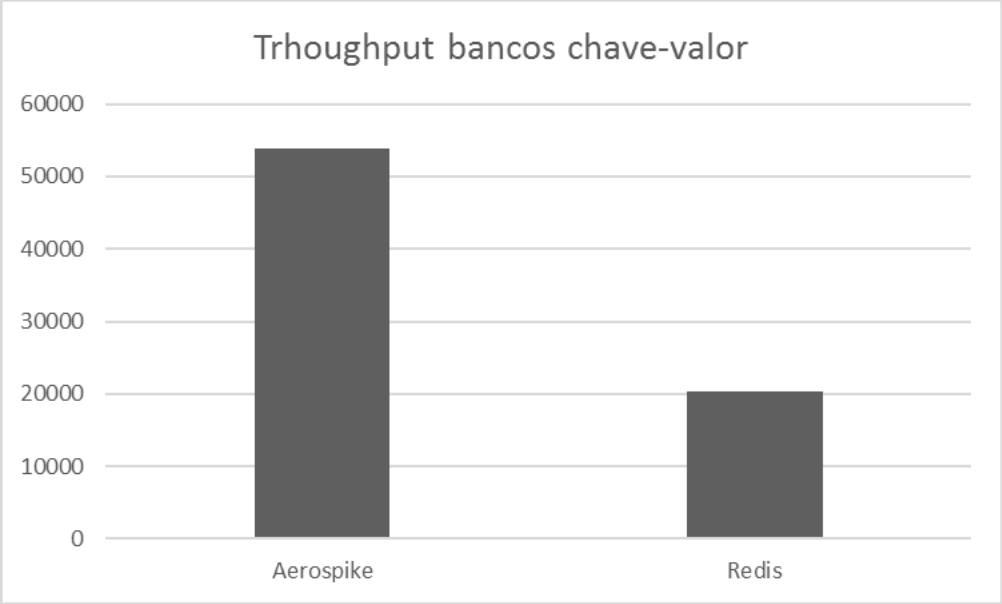
O Teste-T pareado compara as médias de dois grupos entre si, para validar hipóteses (GRAY, 2016). Quando a Sig. for maior que que 0,05, indica que há significância estatística, ou seja, as médias se equivalem estatisticamente, caso o resultado obtido seja inferior, então as médias não são equivalentes.

Quadro 19 – Teste-T de amostras emparelhadas do *Throughput dos* bancos chave-valor

Diferenças emparelhadas							
			95% Intervalo de Confiança da Diferença				
Média	Desvio Padrão	Erro padrão da média	Inferior	Superior	t	df	Sig. (2 extremidades)
-33470,8702	1493,4124	264,0005	-34009,3028	-32932,4375	-126,783	31	,000

O Quadro 19 apresenta o resultado do Teste-T, analisando resultado da Sig., é possível concluir que os *Ths* dos bancos de dados Redis e Aerospike não são iguais, rejeitando assim a hipótese nula. A hipótese alternativa verdadeira é H_2, que aponta o *Th* do Aerospike como sendo maior que o *Th* do Redis, conforme pode ser observado na Figura 14.

Figura 14 – Média de *throughput* de Redis e Aerospike



3.6.1.2 Latência bancos chave-valor

A métrica de latência foi avaliada com base nas médias disponibilizadas pelo *output* do *benchmark*. Foram obtidas 2 amostras de dados para cada iteração, sendo elas: Latência média de leitura e latência média de gravação.

Quanto a latência geral de leitura, foi realizada a análise de normalidade das amostras, e se observou que não há homogeneidade, essa conclusão existe pelo fato do Sig. ser menor do que 0,05, como pode ser observado no Quadro 20.

Quadro 20 - Teste de normalidade da latência de leitura dos bancos chave-valor

Testes de Normalidade						
	Kolmogorov-Smirnov[a]			Shapiro-Wilk		
	Estatística	df	Sig.	Estatística	df	Sig.
Latência de Leitura Redis	,110	32	,200*	,970	32	,504
Latência de Leitura Aerospike	,181	32	,009	,929	32	,037

Quando a significância, é menor do que 0,05, é indicado que que se aplique um teste não paramétrico, nesse caso o teste indicado é Wilcoxon, que conforme descrito no item 2.3.3.7 é uma versão não-paramétrica do Teste-T. O Quadro 21, apresenta o resultado obtido para o teste não paramétrico.

Quadro 21 - Teste Wilcoxon da latência de leitura dos bancos chave-valor

Estatísticas de teste[a]	
	Latência de Leitura Aerospike - Latência de Leitura Redis
Z	-4,937[b]
Significância Sig. (bilateral)	,000

Com a Sig. obtida de 0,000, é possível concluir que a hipótese nula $H_0 - Lt_a = Lt_b$, foi refutada, uma vez que a latência do Redis (*a)* é diferente da latência do Aerospike(*b*). A hipótese alternativa verdadeira é a $H_1 - Lt_a > Lt_b$, onde está afirmado que a latência de leitura do Redis é maior que a latência de leitura do Aerospike, conforme Quadro 22.

Quadro 22 - Médias de latência de leitura dos bancos chave-valor

	Latência de Leitura Redis	Latência de Leitura Aerospike
Média	767,947026933691900	412,996532322514550
N	32	32
Erro Desvio	32,089209574941200	8,907110110986636

Já em relação as médias gerais de latência de inserção, foi executado o teste de normalidade, obteve-se Sig. > que 0,05 para Shapiro-Wilko e Sig. < 0,05 para Kolmogorov-Smirnov, conforme pode ser observado no Quadro 23. Sendo assim foi aplicado os dois modelos de teste, Teste -T (paramétrico) e Teste de Wilcoxon(não paramétrico).

Quadro 23 - Teste de normalidade das médias de latência de gravação dos bancos chave-valor

Testes de Normalidade						
	Kolmogorov-Smirnov[a]			Shapiro-Wilk		
	Estatística	df	Sig.	Estatística	df	Sig.
Latência de Gravação Redis	,107	32	,200*	,967	32	,426
Latência de Gravação Aerospike	,169	32	,021	,939	32	,070

Para o Teste – T e para o Wilcoxon se obteve Sig. igual a 0, sendo assim refutando a hipótese nula, a qual afirma que as latências dos bancos de dados chave-valor homologados são iguais, conforme pode ser observado no Quadro 24.

Quadro 24 - Teste de pares da latência de gravação dos bancos chave-valor

Sig. Teste – T	Sig. Wilcoxon
0,000	0,0000

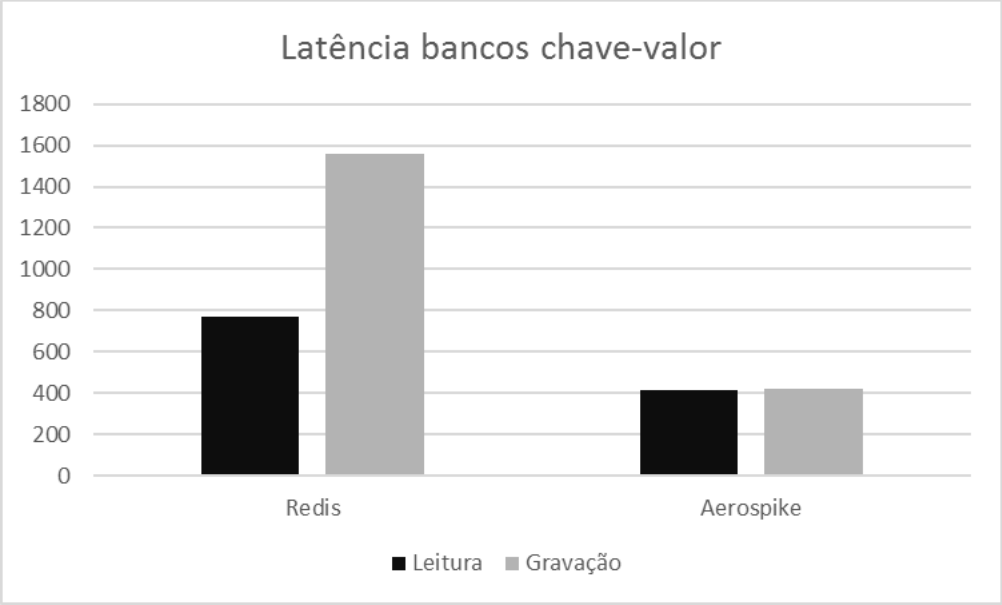
A hipótese verdadeira nesse caso é novamente H_1, pois a latência média de gravação do Redis (*a*) é maior que a latência do Aerospike (*b*), conforme pode ser observado no Quadro 25.

Quadro 25 – Médias da latência de gravação dos bancos chave-valor

	Latência de Gravação Redis	Latência de Gravação Aerospike
Média	1556,456934192544800	422,580340637373470
N	32	32
Erro Desvio	63,814595599169344	9,430177940775515

Com base nas informações de médias obtidas para latência de leitura e gravação, gerou-se o gráfico apresentado na Figura 15, com o objetivo de facilitar a compreensão do resultado obtido.

Figura 15 - Latência de gravação e leitura dos bancos chave-valor



Nesse gráfico é possível observar que existe uma paridade na média da latência das operações do Aerospike, a qual não é observada no banco de dados Redis.

3.6.1.3 RunTime bancos chave-valor

A métrica de *RunTime (Rt)* ou tempo de execução, computa em milissegundos o tempo que foi necessário para que todo o fluxo do *workload* fosse completado. Com os dados obtidos das 32 iterações, foi possível observar que existe significância (Sig. >0,05) no teste de normalidade, o que representa que existe homogeneidade nas amostras, conforme pode ser observado no Quadro 26.

Quadro 26 - Teste de normalidade do *RunTime* dos bancos chave-valor

Testes de Normalidade						
	Kolmogorov-Smirnov[a]			Shapiro-Wilk		
	Estatística	df	Sig.	Estatística	df	Sig.
RunTime Redis	,110	32	,200*	,967	32	,417
RunTime Aerospike	,138	32	,124	,957	32	,234

Havendo homogeneidade foi aplicado o Teste-T pareado, onde se obteve Sig equivalente a 0,000, o que refuta a hipótese H_0, a qual afirma que o tempo de execução de *a* Redis é igual ao tempo de execução de *b* Aerospike, o resultado deste teste encontra-se no Quadro 27.

Quadro 27 - Teste – T do *RunTime* dos bancos chave-valor

Teste de amostras emparelhadas								
	Diferenças emparelhadas					t	df	Sig. (2 extremidades)
	Média	Erro Desvio	Erro padrão da média	95% Intervalo de Confiança da Diferença				
				Inferior	Superior			
RunTime Redis - *RunTime* Aerospike	153934,6	10370,28	1833,224	150195,7	157673,5	83,969	31	0

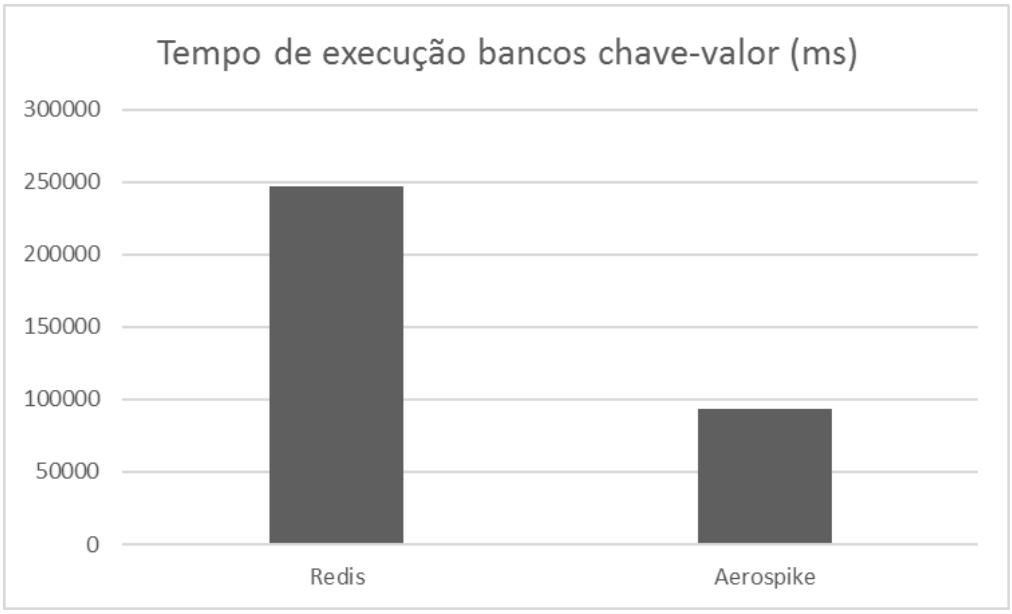
Com base nas médias obtidas é possível corroborar a hipótese H_1, a qual afirma que o *Rt* de *a* é maior que o *Rt* de *b*, sendo assim a média do tempo de execução do Redis, é maior que a média do tempo de execução do Aerospike, conforme pode ser acompanhado no Quadro 28.

Quadro 28 – Estatísticas de amostras emparelhadas *RunTime* dos bancos chave-valor

Estatísticas de amostras emparelhadas				
	Média	N	Erro Desvio	Erro padrão da média
RunTime Redis	247016,2	32	10212,99	1805,419
RunTime Aerospike	93081,56	32	2216,231	391,778

A fim de facilitar a visualização do resultado obtido, gerou-se uma representação gráfica das médias, a qual está apresentada na Figura 16.

Figura 16 - *RunTime* bancos chave-valor



3.6.1.4 Throughput dos bancos orientados a documentos

O *Throughput* representa o número de operações por segundo que o banco de dados foi capaz de processar durante o tempo de execução do teste. Neste caso foi coletada uma amostra de 32 iterações. Para avaliar qual modelo de teste deveria ser aplicado, se paramétrico ou não paramétrico, foi realizado o teste de normalidade para avaliar a homogeneidade das amostras, o resultado desse teste está apresentado no Quadro 29.

Quadro 29 – Testes de Normalidade do *Throughput* dos bancos orientados a documentos

Testes de Normalidade						
	Kolmogorov-Smirnov[a]			Shapiro-Wilk		
	Estatística	df	Sig.	Estatística	df	Sig.
MongoDB *Throughput*	,130	32	,183	,977	32	,697
Couchbase *Throughput*	,084	32	,200*	,977	32	,705

Avaliando s significância apresentada no teste de normalidade, é possível concluir que no caso da métrica de *Throughput* existe homogeneidade nas amostras, sendo assim o teste de pares recomendado é o Teste – T.

Quadro 30 – Teste – T do *Throughput* dos bancos orientados a documentos

Teste de amostras emparelhadas							
Diferenças emparelhadas					t	df	Sig. (2 extremidades)
Média	Erro Desvio	Erro padrão da média	95% Intervalo de Confiança da Diferença				
			Inferior	Superior			

-1134,61	61,11	10,80	-1156,65	-1112,58	-105,02	31,00	0,00

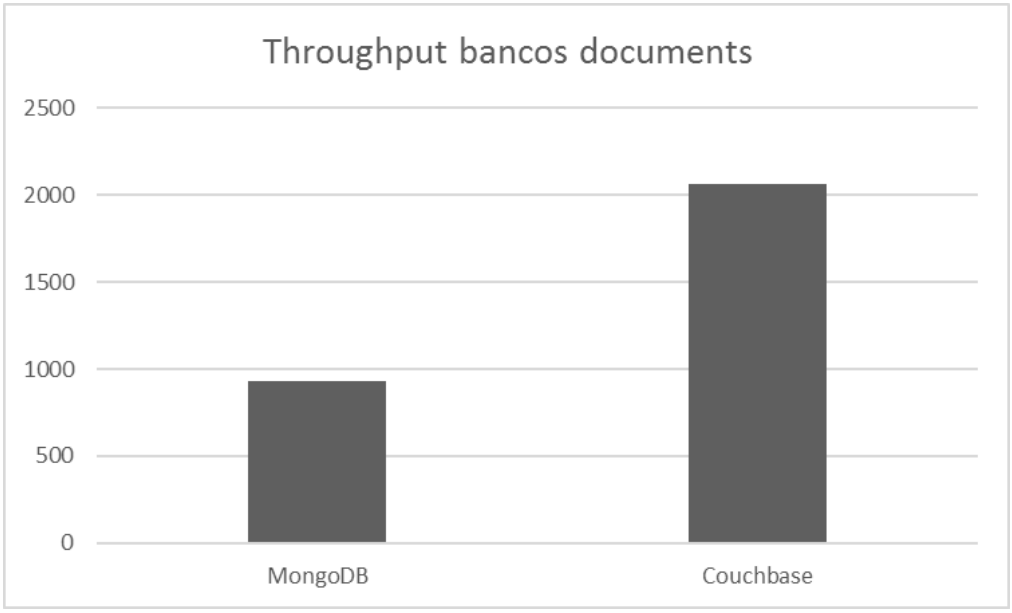
Como pode ser observado no Quadro 30, o par não possui Sig. maior que 0,05, sendo assim, pode ser refutada a hipótese nula $H_0 - Th_a = Th_b$, onde está afirmado que o *Throughput* do MongoDB (*a*) é igual ao *Throughput* do Couchbase (*b*).

Quadro 31 – Médias de *Throughput* dos bancos orientados a documentos

Estatísticas de amostras emparelhadas				
	Média	N	Erro Desvio	Erro padrão da média
MongoDB *Throughput*	928,25	32,00	45,59	8,06
Couchbase *Throughput*	2062,87	32,00	33,15	5,86

Nesse caso, a hipótese alternativa que se confirma é $H_2 - Th_a < Th_b$, pois, conforme apresentado no a média do *Throughput* do MongoDB (*a*) é menor que a média do *Throughput* do Couchbase (*b*), conforme apresentado no Quadro 31, e representado graficamente na Figura 17.

Figura 17 - *Throughput* dos bancos orientados a documentos



3.6.1.5 Latência bancos orientados a documentos

A métrica de latência dos bancos de dados orientados a documentos foi avaliada com base nas médias disponibilizadas pelo *output* do *benchmark*. Foram obtidas 2 amostras de dados para cada iteração, sendo elas: Latência média de leitura e latência média de gravação.

Quanto a latência geral de leitura, foi realizada a análise de normalidade das amostras, o resultado obtido está apresentado no Quadro 32. Com este resultado é

possível concluir, que não existe homogeneidade na amostra do MongoDB, sendo assim o teste de pares que deve ser aplicado é Wilcoxon.

Quadro 32 – Teste de normalidade de latência de leitura dos bancos orientados a documentos

Testes de Normalidade						
	Kolmogorov-Smirnov[a]			Shapiro-Wilk		
	Estatística	df	Sig.	Estatística	df	Sig.
MongoDB Latência de Leitura	,193	32	,004	,933	32	,046
Couchbase Latência de Leitura	,084	32	,200*	,982	32	,846

O resultado obtido com o teste de Wilcoxon, está apresentado no Quadro 33, como o esta Sig sendo menor que 0,05, é possível afirmar que a hipótese nula apresentada anteriormente, a qual afirma que os bancos de dados MongoDB e Couchbase são iguais, está refutada.

Quadro 33 – Teste de Wilcoxon da latência de leitura dos bancos orientados a documentos

Estatísticas de teste[a]	
	Couchbase Latência de Leitura MongoDB Latência de Leitura
Z	-4,937[b]
Significância Sig. (bilateral)	,000

Estando refutada a hipótese nula, com base nas médias obtidas e apresentadas no Quadro 34, é possível afirmar que a hipótese alternativa verdadeira é $H_1 - Lt_a > Lt_b$, a qual afirma que a latência de leitura do MongoDB (*a*) é maior que a latência de leitura do Couchbase (*b*).

Quadro 34 – Médias da latência de leitura dos bancos orientados a documentos

	MongoDB Latência de Leitura	Couchbase Latência de Leitura
Média	11729,034086773941	6035,439042032758
N	32	32
Erro Desvio	790,921275495594	104,62852702148878

Quanto a métrica de latência de gravação, o Quadro 35 apresenta o teste de normalidade, o qual indica que o teste aplicado deve ser não paramétrico, em função do Sig ser menor do que 0,05, sendo assim o teste executado foi Wilcoxon.

Quadro 35 – Teste de normalidade da latência de gravação dos bancos orientados a documentos

Testes de Normalidade						
	Kolmogorov-Smirnov[a]			Shapiro-Wilk		
	Estatística	df	Sig.	Estatística	df	Sig.
MongoDB Latência de Gravação	,125	32	,200*	,943	32	,093
Couchbase Latência de Gravação	,149	32	,067	,924	32	,027

Como pode ser observado no Quadro 36, o par não possui Sig. maior que 0,05, sendo assim, pode ser refutada a hipótese nula $H_0 - Lt_a = Lt_b$, onde está afirmado que a latência de gravação do MongoDB (*a*) é igual a latência de gravação do Couchbase (*b*).

Quadro 36 – Teste de Wilcoxon da latência de gravação dos bancos orientados a documentos

Estatísticas de teste[a]	
	Couchbase Latência de Gravação MongoDB Latência de Gravação
Z	-4,937[b]
Significância Sig. (bilateral)	,000

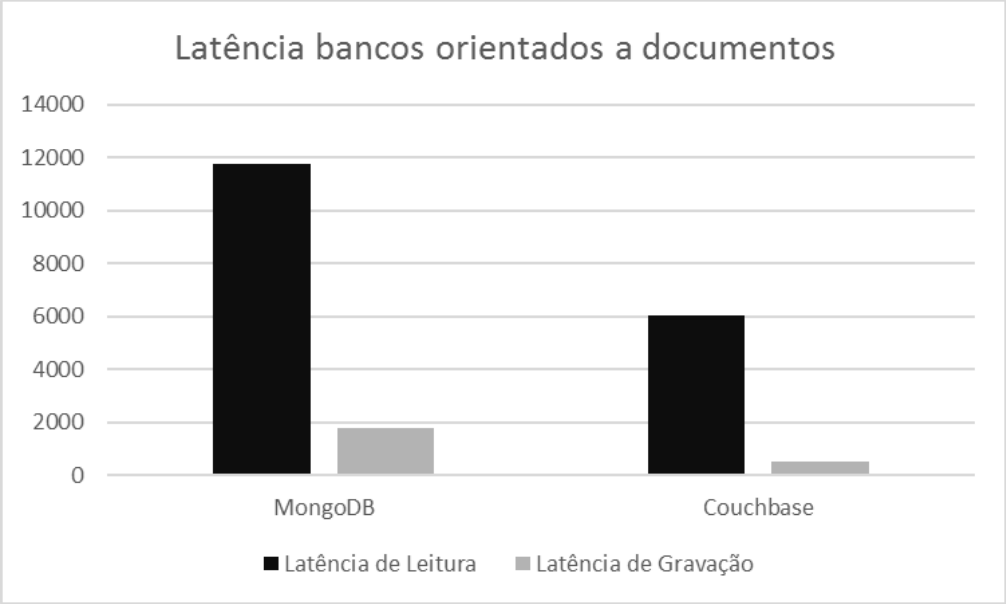
Não havendo significância, a hipótese alternativa que se confirma é $H_1 - Lt_a > Lt_b$, pois, conforme apresentado no Quadro 37, a média da latência de gravação do MongoDB (*a*) é maior que a latência de gravação do Couchbase (*b*).

Quadro 37 – Médias da latência de gravação dos bancos orientados a documentos

	MongoDB Latência de Gravação	Couchbase Latência de Gravação
Média	1793,8624465193136	512,6429485399568
N	32	32
Erro Desvio	122,341173765936220	5,286435946635227

Com base nas informações de médias obtidas para latência de leitura e gravação, gerou-se o gráfico apresentado na Figura 18 com o objetivo de facilitar a compreensão do resultado obtido.

Figura 18 - Latência de gravação e leitura bancos orientados a documentos



3.6.1.6 RunTime bancos orientados a documentos

A métrica de *RunTime (Rt)* ou tempo de execução, computa em milissegundos o tempo que foi necessário para que todo o fluxo do *workload* fosse completado. Com os dados obtidos das 32 iterações. Para avaliar qual modelo de teste deveria ser aplicado, se paramétrico ou não paramétrico, foi realizado o teste de normalidade para avaliar a homogeneidade das amostras, o resultado desse teste está apresentado no, Quadro 38.

Quadro 38 – Teste de normalidade bancos orientados a documentos

Testes de Normalidade						
	Kolmogorov-Smirnov[a]			Shapiro-Wilk		
	Estatística	df	Sig.	Estatística	df	Sig.
MongoDB *RunTime*	0,357	32	0,000	0,466	32	0,000
Couchbase *RunTime*	0,085	32	,200*	0,978	32	0,750

Conforme observado não existe homogeneidade nas amostras do MongoDB, sendo assim o teste que deve ser aplicado nesse caso é o teste não paramétrico Wilcoxon.

Quadro 39 – Teste de Wilcoxon bancos orientados a documentos

Estatísticas de teste[a]	
	Couchbase *RunTime* - MongoDB *RunTime*
Z	-4,880[b]
Significância Sig. (bilateral)	,000

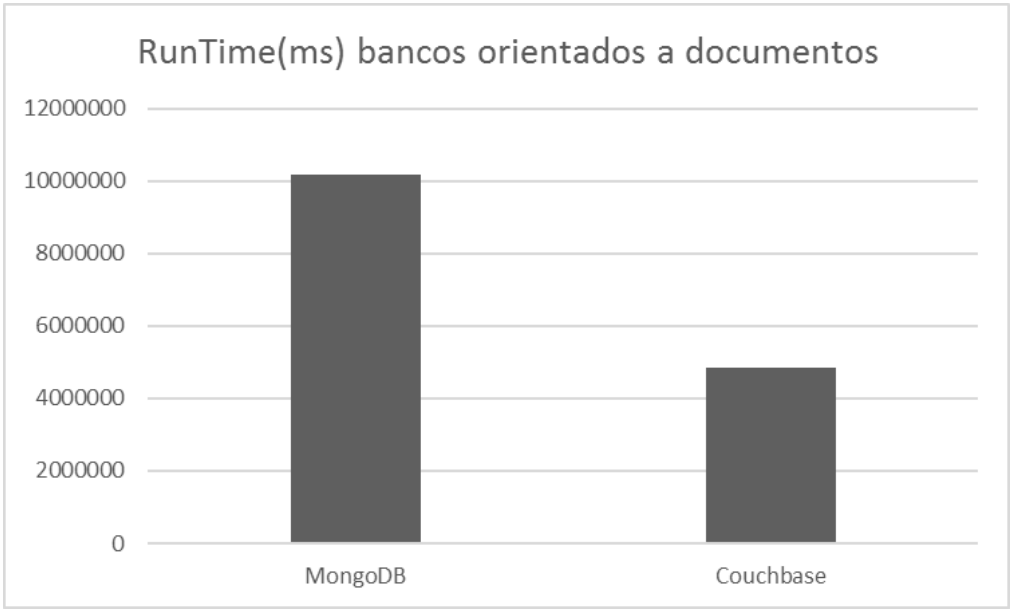
Como pode ser observado no Quadro 39, o par não possui Sig. maior que 0,05, sendo assim pode ser refutada a hipótese nula $H_0 - Rt_a = Rt_b$, onde está afirmado que o *RunTime* do MongoDB (*a*) é igual ao *RunTime* do Couchbase (*b*). Nesse caso, a hipótese alternativa que se confirma é $H_1 - Rt_a > Rt_b$, pois, conforme apresentado no Quadro 40, a média do *RunTime* do MongoDB (*a*) é maior que o *RunTime* do Couchbase (*b*).

Quadro 40 – Médias *RunTime* bancos orientados a documentos

	MongoDB *RunTime*(ms)	Couchbase *RunTime*(ms)
Média	10189870	4848832
N	32	32
Erro Desvio	2446747	77776,34

A fim de facilitar a visualização do resultado obtido, gerou-se uma representação gráfica das médias, a qual está apresentada na Figura 19.

Figura 19 - RunTime bancos orientados a documentos



3.6.1.7 Throughput Cassandra x Couchbase

O *Throughput* representa o número de operações por segundo que o banco de dados foi capaz de processar durante o tempo de execução do teste. Neste caso foi

coletada uma amostra de 32 iterações. Para avaliar qual modelo de teste deveria ser aplicado, se paramétrico ou não paramétrico, foi realizado o teste de normalidade para avaliar a homogeneidade das amostras, o resultado desse teste está apresentado no Quadro 41.

Quadro 41 – Normalidade *Throughput* dos bancos Cassandra e Couchbase

Testes de Normalidade						
	Kolmogorov-Smirnov[a]			Shapiro-Wilk		
	Estatística	df	Sig.	Estatística	df	Sig.
Cassandra Throughput	,240	32	,000	,751	32	,000
Couchbase Throughput	,084	32	,200*	,977	32	,705

Com base no resultado obtido no teste de normalidade, onde o Sig. resultou em 0, então deve ser aplicado um teste não paramétrico, nesse caso Wilcoxon. Sendo assim, o resultado obtido neste teste está apresentado no Quadro 42.

Quadro 42 – Teste de Wilcoxon *Throughput* dos bancos Cassandra e Couchbase

Estatísticas de teste[a]	
	Cassandra Throughput - Couchbase Throughput
Z	-4,937[b]
Significância Sig. (bilateral)	,000

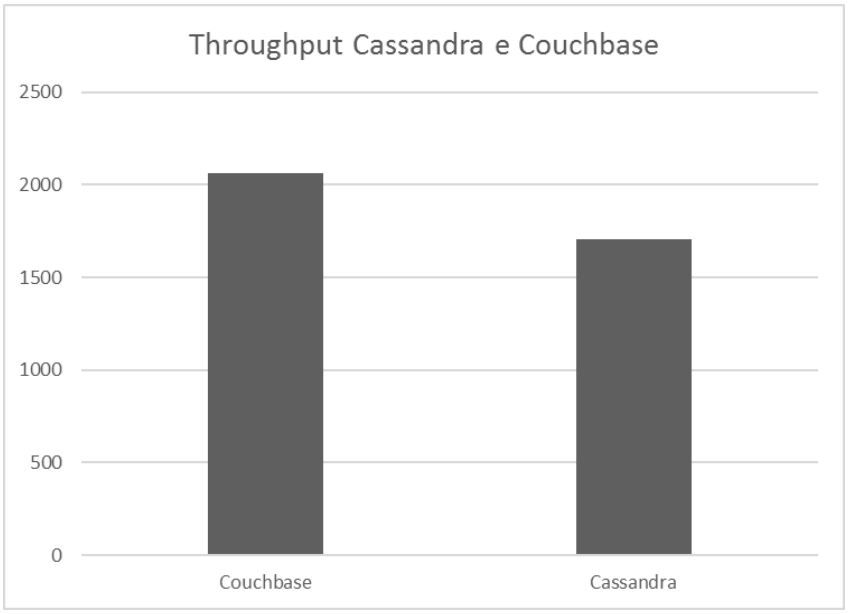
Analisando o resultado é possível refutar a hipótese nula que afirma que o Cassandra e o Couchbase são iguais, sendo assim se corrobora a hipótese alternativa $H_2 - Th_a < Th_b$, a qual afirma que o *throughput* do Cassandra é menor do que o *throughput* do Couchbase. As médias obtidas estão apresentadas no Quadro 43.

Quadro 43 – Médias de *Throughput* dos bancos Cassandra e Couchbase

Relatório		
	Couchbase Throughput	Cassandra Throughput
Média	2062,867371129771000	1707,979158261082800
N	32	32
Erro Desvio	33,150099916753526	30,540498537370752

A fim de facilitar a visualização dos resultados a Figura 20, apresenta graficamente as médias obtidas.

Figura 20 - *Throughput* Cassandra e Couchbase



3.6.1.8 Latência Cassandra x Couchbase

A métrica de latência dos bancos de dados orientados a documentos foi avaliada com base nas médias disponibilizadas pelo *output* do *benchmark*. Foram obtidas 2 amostras de dados para cada iteração, sendo elas: Latência média de leitura e latência média de gravação.

Quanto a latência geral de leitura, foi realizada a análise de normalidade das amostras, o resultado obtido está apresentado no Quadro 44. Com este resultado é possível concluir, que existe homogeneidade nas amostras, sendo assim o teste de pares que deve ser aplicado é Teste-T.

Quadro 44 – Teste de normalidade latência leitura bancos Cassandra e Couchbase

Testes de Normalidade						
	Kolmogorov-Smirnov[a]			Shapiro-Wilk		
	Estatística	df	Sig.	Estatística	df	Sig.
Cassandra Latência Leitura	,116	32	,200*	,966	32	,389
Couchbase Latência de Leitura	,084	32	,200*	,982	32	,846

Aplicando o Teste – T sobre as amostras, se obteve o resultado apresentado no Quadro 45. Tal resultado demonstra que o Sig nesse caso é menor do que 0,05, sendo assim, os bancos Cassandra e Couchbase não possuem significância, refutando a hipótese nula.

Quadro 45 – Teste - T latência de leitura bancos Cassandra e Couchbase

Teste de amostras emparelhadas Latência Leitura								
	Diferenças emparelhadas					t	df	Sig. (2 extremidades)
	Média	Erro Desvio	Erro padrão da média	95% Intervalo de Confiança da Diferença				
				Inferior	Superior			
Couchbase - Cassandra	2559,06	121,92	21,55	2515,10	2603,02	118,73	31,00	0,00

Sendo que a hipótese nula foi refutada, é possível afirmar que a hipótese alternativa verdadeira nesse caso é $H_2 - Lt_a < Lt_b$, a qual afirma que a latência de *a* Cassandra é menor do que a latência de *b* Couchbase. Isso pode ser confirmado se observar as médias apresentadas no Quadro 46.

Quadro 46 – Médias de latência de leitura bancos Cassandra e Couchbase

Estatísticas de amostras emparelhadas				
	Média	N	Erro Desvio	Erro padrão da média
Couchbase Latência de Leitura	6035,439042	32	104,628527	18,49588524
Cassandra Latência Leitura	3476,378792	32	51,63186838	9,127311063

Quanto a métrica de latência de gravação, o Quadro 47 apresenta o teste de normalidade, o qual indica que o teste aplicado deve ser não paramétrico, em função do Sig ser menor do que 0,05, sendo assim o teste executado foi Wilcoxon.

Quadro 47 – Teste de normalidade latência de gravação bancos Cassandra e Couchbase

Testes de Normalidade						
	Kolmogorov-Smirnov[a]			Shapiro-Wilk		
	Estatística	df	Sig.	Estatística	df	Sig.
Cassandra Latência Gravação	,229	32	,000	,671	32	,000
Couchbase Latência de Gravação	,149	32	,067	,924	32	,027

O resultado obtido com o teste de Wilcoxon, está apresentado no Quadro 48, como o esta Sig sendo menor que 0,05, é possível afirmar que a hipótese nula apresentada anteriormente, a qual afirma que os bancos de dados Cassandra e Couchbase são iguais, está refutada.

Quadro 48 – Teste de Wilcoxon da latência de gravação bancos Cassandra e Couchbase

Latência de Gravação	
Z	-4,937b
Significância Sig. (bilateral)	,000

Estando refutada a hipótese nula, com base nas médias obtidas e apresentadas no Quadro 49, é possível afirmar que a hipótese alternativa verdadeira

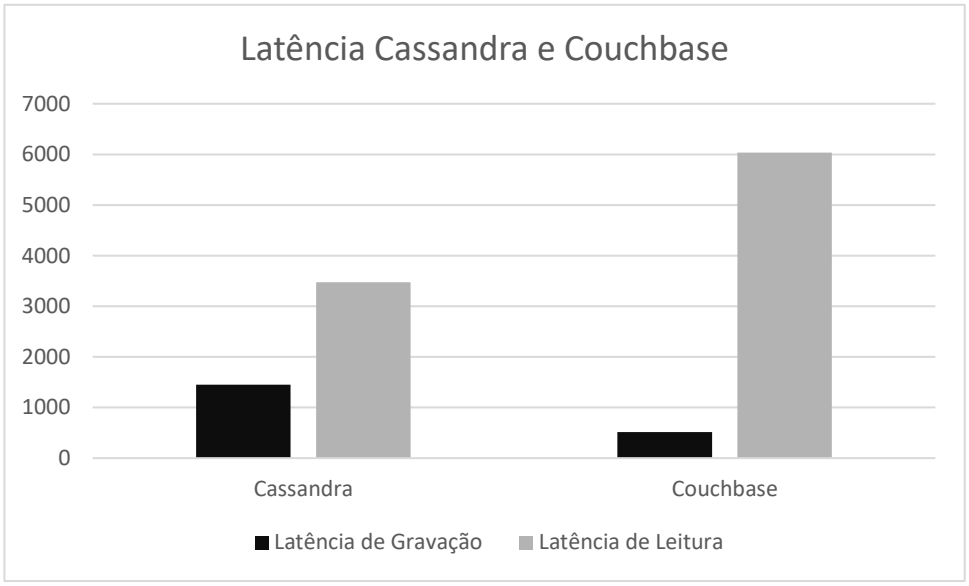
é $H_1 - Lt_a > Lt_b$, a qual afirma que a latência de leitura do MongoDB (*a*) é maior que a latência de leitura do Couchbase (*b*).

Quadro 49 – Médias da latência gravação dos bancos orientados a documentos

	Cassandra	Couchbase
Média	1453,331484076764000	512,642948539956800
N	32	32
Erro Desvio	45,101315592705056	5,286435946635227

Comparando as duas latências graficamente na Figura 21, é possível observar que a latência de gravação do Cassandra é maior que a do Couchbase, já a latência de leitura do Couchbase é maior do que a do Cassandra.

Figura 21 - Latência Cassandra e Couchbase



3.6.1.9 RunTime Cassandra e Couchbase

A métrica de *RunTime (Rt)* ou tempo de execução, computa em milissegundos o tempo que foi necessário para que todo o fluxo do *workload* fosse completado. Com os dados obtidos das 32 iterações. Para avaliar qual modelo de teste deveria ser aplicado, se paramétrico ou não paramétrico, foi realizado o teste de normalidade para avaliar a homogeneidade das amostras, o resultado desse teste está apresentado no, Quadro 50.

Quadro 50 – Teste de normalidade RunTime Cassandra e Couchbase

Testes de Normalidade						
	Kolmogorov-Smirnov[a]			Shapiro-Wilk		
	Estatística	df	Sig.	Estatística	df	Sig.
Cassandra RunTime	,250	32	,000	,723	32	,000
Couchbase RunTime	,085	32	,200*	,978	32	,750

Conforme observado não existe homogeneidade nas amostras do Cassandra, sendo assim o teste que deve ser aplicado nesse caso é o teste não paramétrico Wilcoxon.

Quadro 51 – Wilcoxon RunTime Cassandra e Couchbase

RunTime	
	Couchbase - Cassandra
Z	-4,937b
Significância Sig. (bilateral)	,000

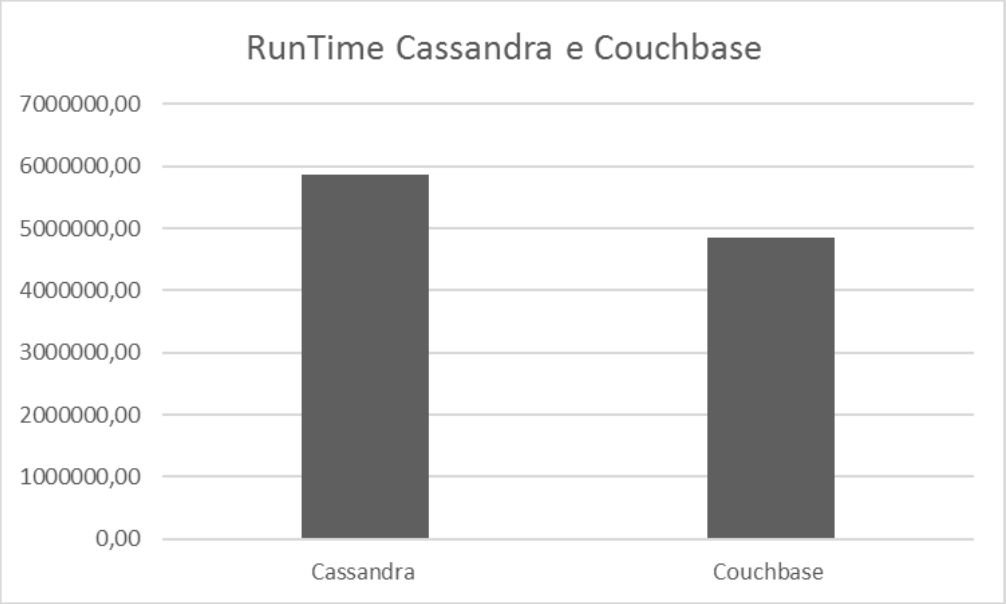
Como pode ser observado no Quadro 51, o par não possui Sig. maior que 0,05, sendo assim pode ser refutada a hipótese nula $H_0 - Rt_a = Rt_b$, onde está afirmado que o *RunTime* do Cassandra (*a*) é igual ao *RunTime* do Couchbase (*b*). Nesse caso, a hipótese alternativa que se confirma é $H_1 - Rt_a > Rt_b$, pois, conforme apresentado no Quadro 52, a média do *RunTime* do Cassandra (*a*) é maior que o *RunTime* do Couchbase (*b*).

Quadro 52 – Médias *RunTime* Cassandra e Couchbase

	Cassandra RunTime	Couchbase RunTime
Média	5856773,97	4848831,81
N	32,00	32,00
Erro Desvio	109844,05	77776,34

A fim de facilitar a observação dos resultados, as médias estão expostas graficamente na Figura 22.

Figura 22 - RunTime Cassandra e Couchbase



3.6.2 Profile

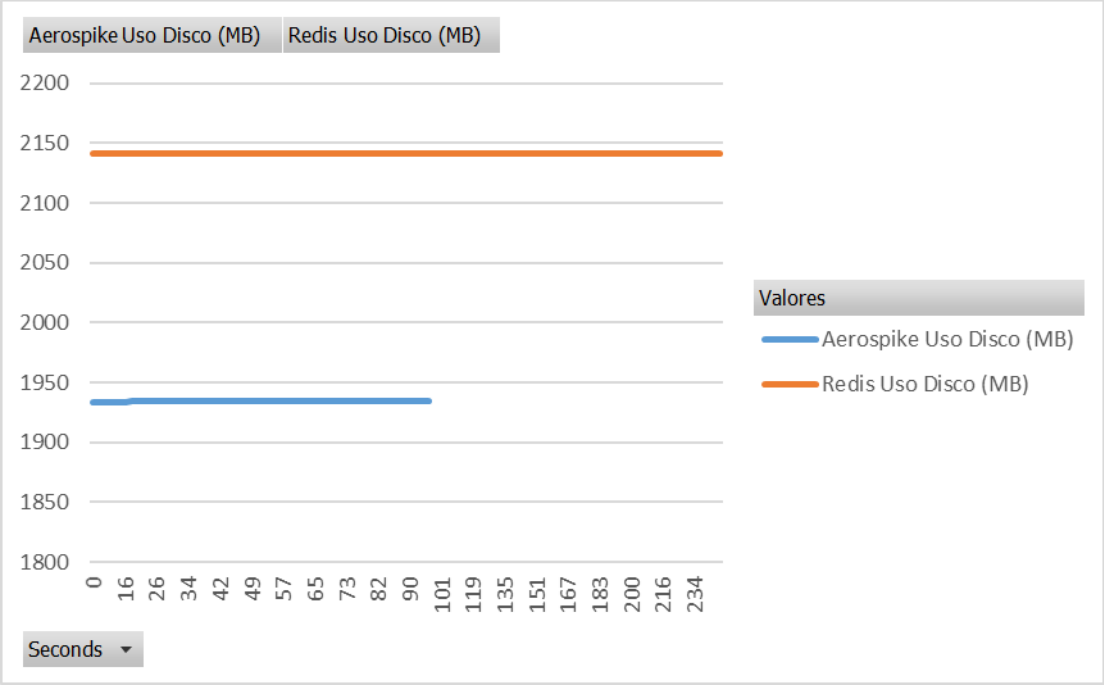
Nesta seção estão apresentadas as análises sobre o desempenho e a forma de utilização dos recursos computacionais de cada banco de dados, cujos dados foram obtidos com o monitoramento do servidor de banco e com a saída do *benchmark* YCSB. No total foram executadas 32 iterações de cada um dos testes de carga, porém para estruturar o *profile* dos bancos foram utilizados apenas os dados da 16ª iteração.

3.6.2.1 Bancos chave-valor

Conforme já citado anteriormente, a carga de dados executada para os bancos de dados chave-valor teve um tamanho de 5GB, essa carga realizou operações de inserção e leitura dos dados, sendo que foi executado 50% de cada uma das operações.

A Figura 23 apresenta a utilização do espaço em disco dos bancos de dados chave-valor Redis e do Aerospike, sendo que eixo X representa o tempo de execução do teste e já o eixo Y o espaço em disco utilizado, observando que no total o espaço disponível é de 20GB.

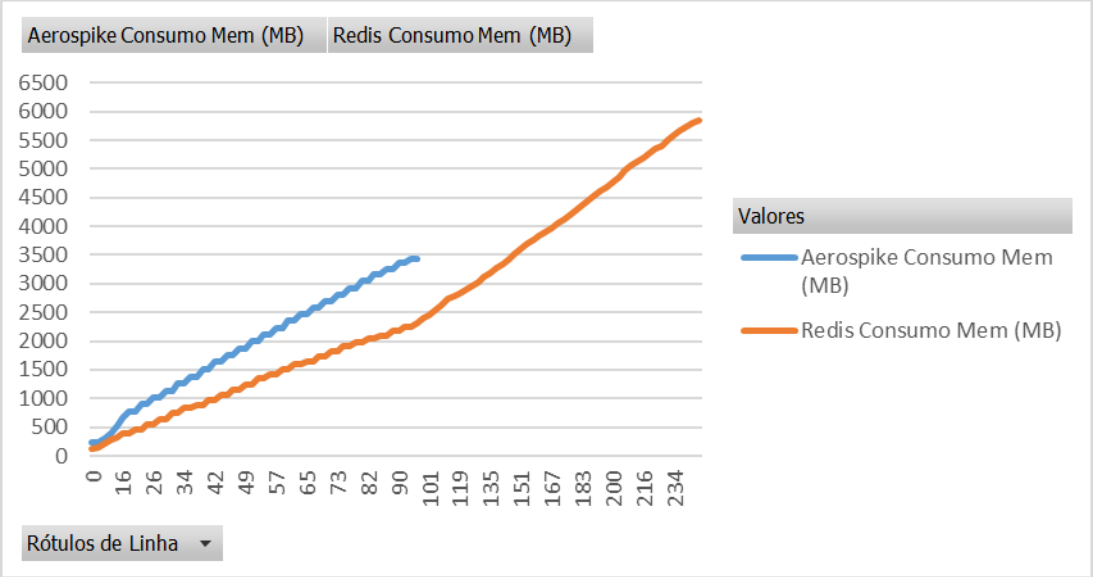
Figura 23 - Consumo de disco de Aerospike e Redis



Como é possível observar na Figura 23, durante a execução do teste não houve alteração na utilização do espaço em disco, isso ocorre porque os testes foram realizados sem persistência em disco, como explicado na seção 3.5.3.3, e o processamento dos dados ocorreu todo na memória RAM. Observou-se também que não houve a necessidade de o sistema operacional realizar *swap* para concluir a tarefa.

Outro recurso de *hardware* avaliado foi o consumo de memória RAM, mantendo a carga já descrita acima e referente à 16ª iteração. O servidor possuía 16GB de memória, porém, para facilitar a visualização o gráfico foi limitado a 8GB. Pode-se observar na Figura 24 as diferenças na gestão da memória RAM dos bancos em questão para a mesma carga de dados.

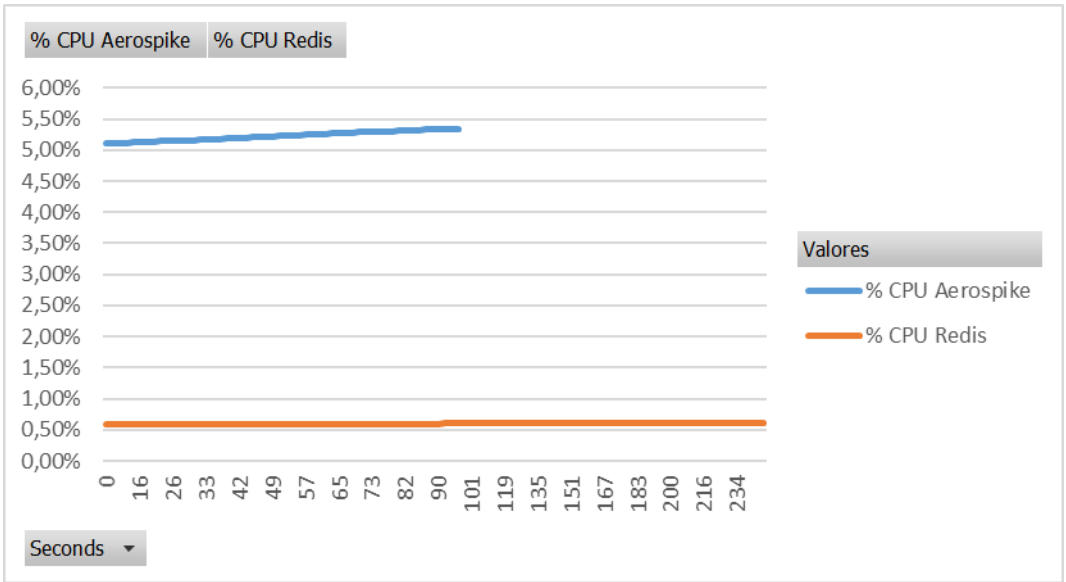
Figura 24 - Consumo de memória RAM de Aerospike e Redis



Ao final da execução o Aerospike utiliza menos espaço de memória para armazenar o mesmo volume de dados que o Redis ao fim de sua execução. O consumo de memória do Aerospike é linear, conforme previsto na documentação disponível em http://www.aerospike.com/docs/operations/plan/capacity/ (AEROSPIKE, 2012).

Quanto à utilização de CPU do servidor de banco de dados, é possível observar na Figura 25 que o consumo deste recurso por parte do Aerospike é maior do que o Redis, porém ambos bancos de dados realizaram o trabalho com um consumo muito baixo de processamento do servidor durante toda a execução do *workload*.

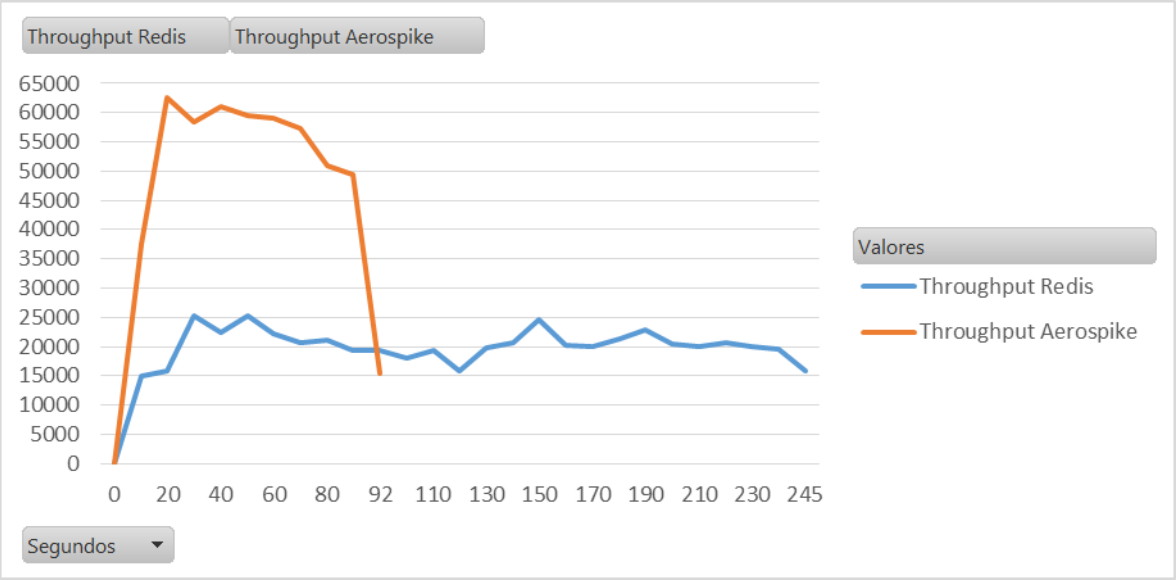
Figura 25 - Consumo de CPU de Aerospike e Redis



De acordo com Aerospike (2012), este banco de dados não possui uma forte dependência do desempenho da CPU. Devido à implementação simples que ele utiliza para realizar as requisições *get* e *set*, as CPUs raramente são estressadas pelo processo do banco de dados. Já o consumo de CPU tão inferior do Redis é explicado, pelo fato de o mesmo ser *single threaded* (REDIS, 2009).

A métrica de *througuput* também foi avaliada na iteração 16, a fim de observar o comportamento dos bancos durante a execução da carga. A Figura 26 apresenta graficamente a evolução das operações por segundo de cada um dos bancos de dados.

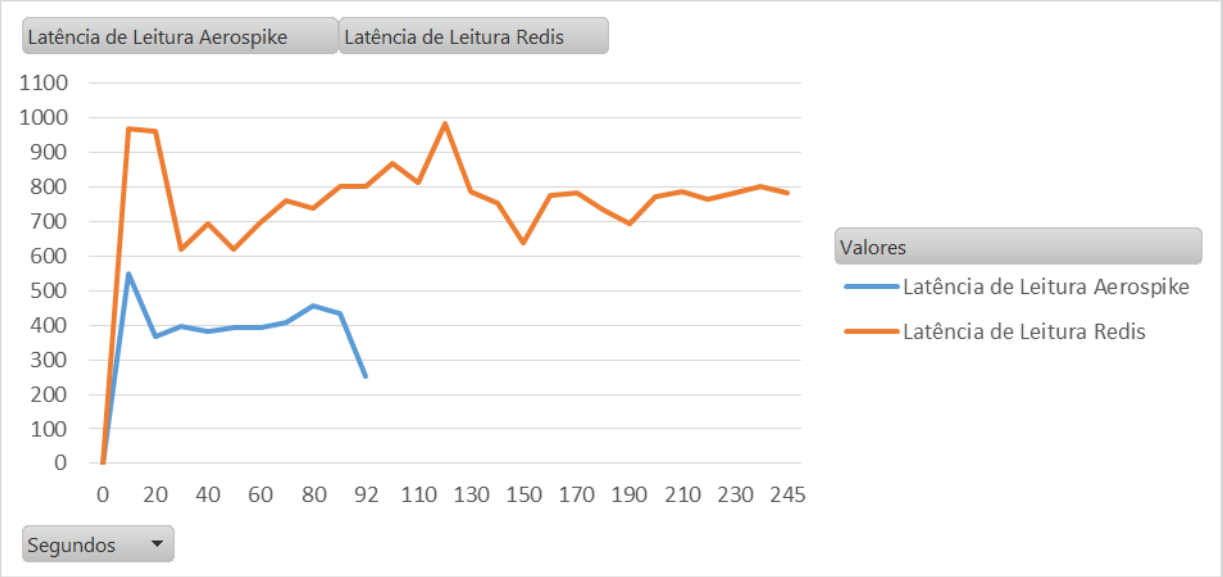
Figura 26 - *Throughput* de Aerospike e Redis



Conforme já observado na análise estatística o Aeropike possui uma média de *throughput* maior que o Redis, isso pode ser melhor visualizado quando apresentada a evolução desta métrica durante o tempo de execução, a métricas foram coletadas de 10 em 10 segundos, e aos 20 segundos o Aeropike chegou a atingir 62526,2 ops/sec, sendo que o pico do Redis foi de 25277,27.

Assim como as demais medidas a latência de leitura durante a execução da iteração 16, está apresentada na Figura 27. Essa informação se refere ao tempo de processamento da operação de leitura do cliente, processamento no servidor e retorno da informação para o cliente.

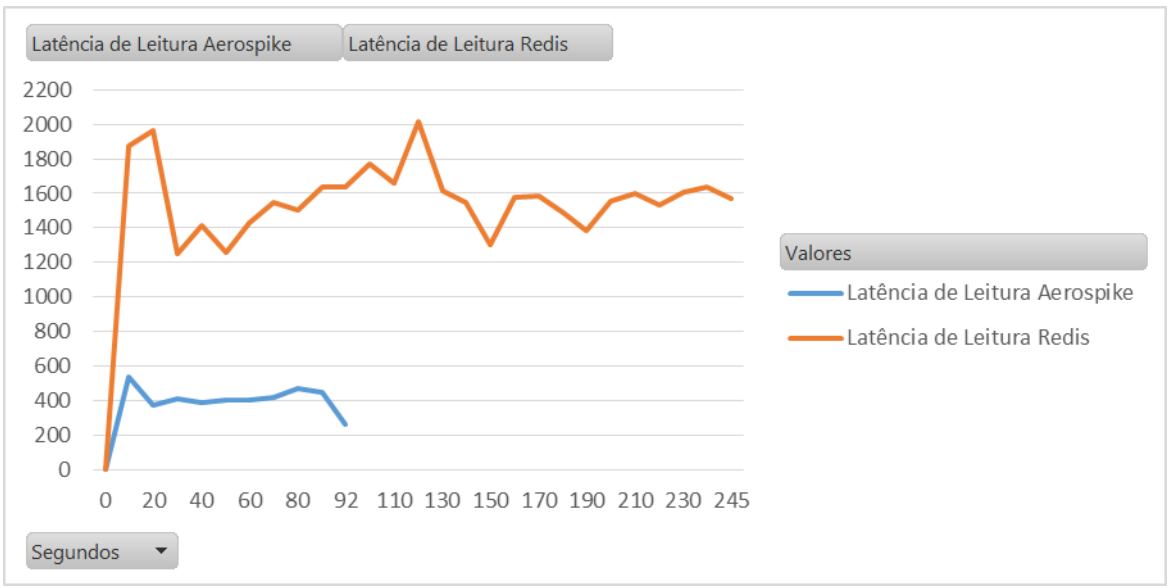
Figura 27 – Latência de leitura de Aerospike e Redis



De acordo com o resultado obtido na análise estatística, a média da latência de leitura do Redis é maior do que a do Aerospike, isso se reafirma se observado o gráfico apresentado. Além disso durante a execução da carga, a latência do Aerospike se mantém mais estável do que a do Redis.

Do mesmo modo que é importante apresentar os resultados da latência de leitura, é necessário apresentar os resultados da latência de inserção, os quais podem ser visualizados na Figura 28. Essa informação se refere ao tempo de processamento da operação de inserção do cliente, processamento no servidor e retorno da informação para o cliente.

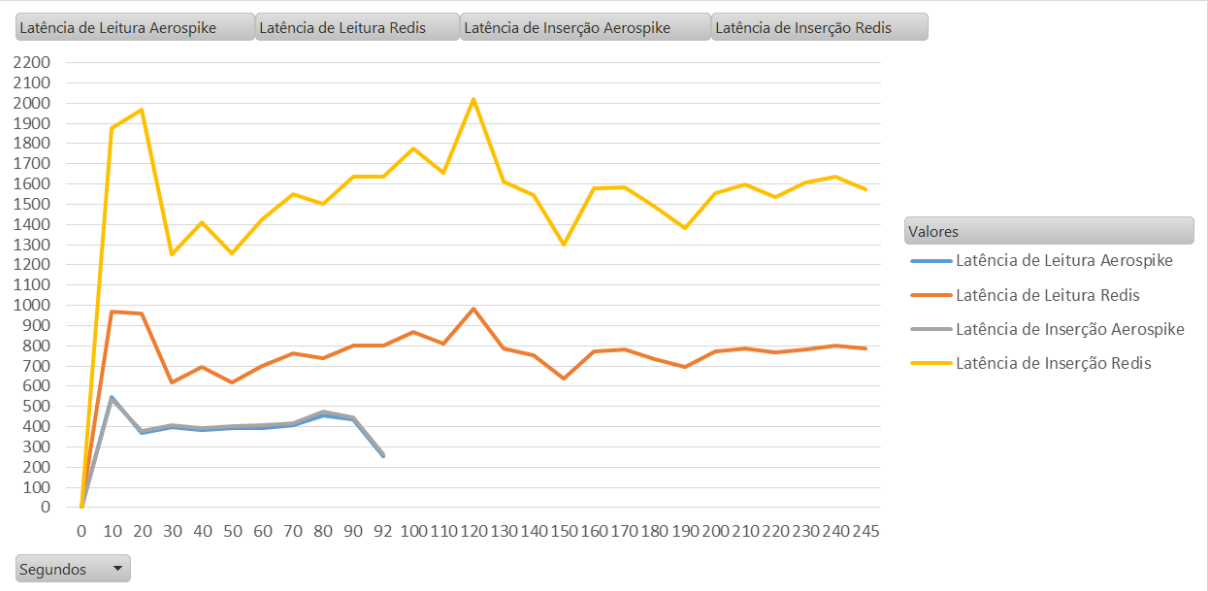
Figura 28 – Latência de inserção de Aerospike e Redis



Neste resultado, nota-se que o Aerospike na métrica de latência de inserção mantém sua linha estável, enquanto o Redis aumenta consideravelmente o tempo de resposta para as operações de inserção, sendo que a diferença ultrapassa em alguns pontos 1000us.

Sintetizando as informações de latência em um único gráfico, apresentado na Figura 29, nota-se que o Redis praticamente dobra a latência nas operações de inserção se comparado com suas operações de leitura. Já o Aerospike mantém um comportamento estável em ambas as operações.

Figura 29 – Latência (Leitura e Gravação) de Aerospike e Redis

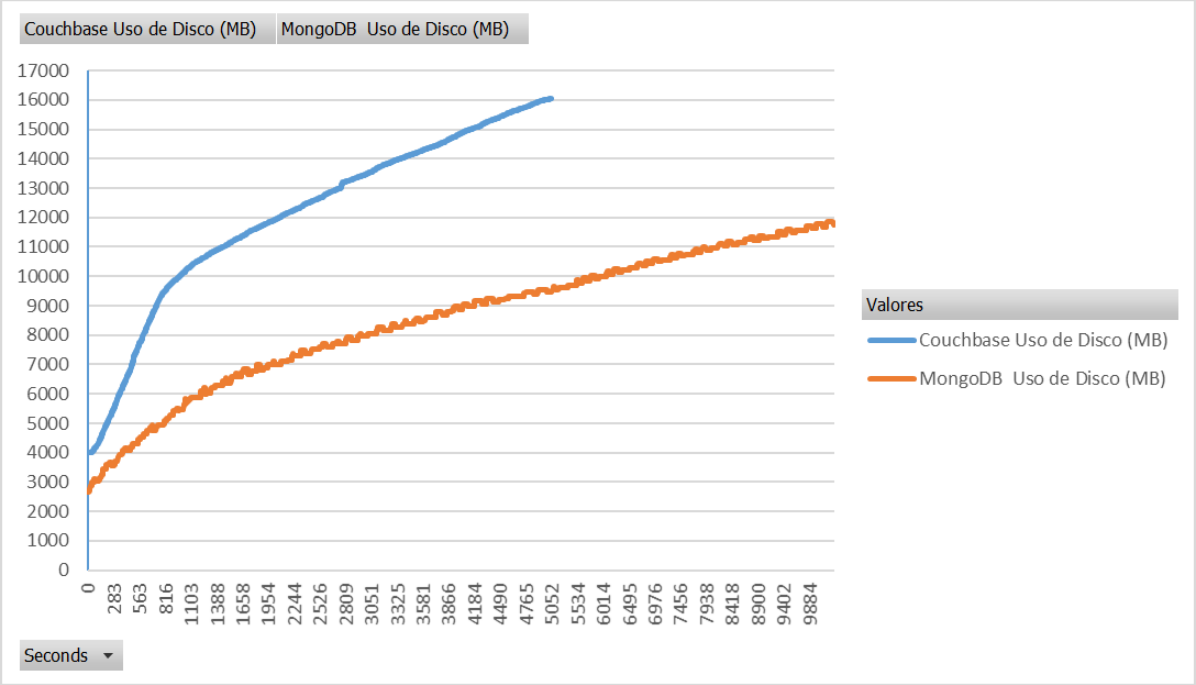


3.6.2.2 Bancos orientados a documentos

Conforme já citado anteriormente, a carga de dados executada para os bancos de dados orientados a documentos teve um tamanho de 10GB, essa carga realizou operações de inserção e leitura dos dados, sendo que foi executado 75% foram inserções e 25% leitura.

A Figura 30 apresenta a utilização do espaço em disco dos bancos de dados *documents* MongoDB e do Couchbase, sendo que eixo X representa o tempo de execução do teste e o eixo Y o espaço em disco utilizado, observando que o total do espaço disponível é de 20GB.

Figura 30 - Consumo de disco de Couchbase e MongoDB



É possível observar que o Couchbase ocupa mais espaço em disco para armazenamento dos dados do que o MongoDB e possui um crescimento mais acentuado. Isto é explicado pelo fato de o *storage engine* WiredTiger utilizado no MongoDB possuir habilitado por padrão o mecanismo de compressão Snappy, que compacta e descompacta os dados de forma transparente ao usuário.

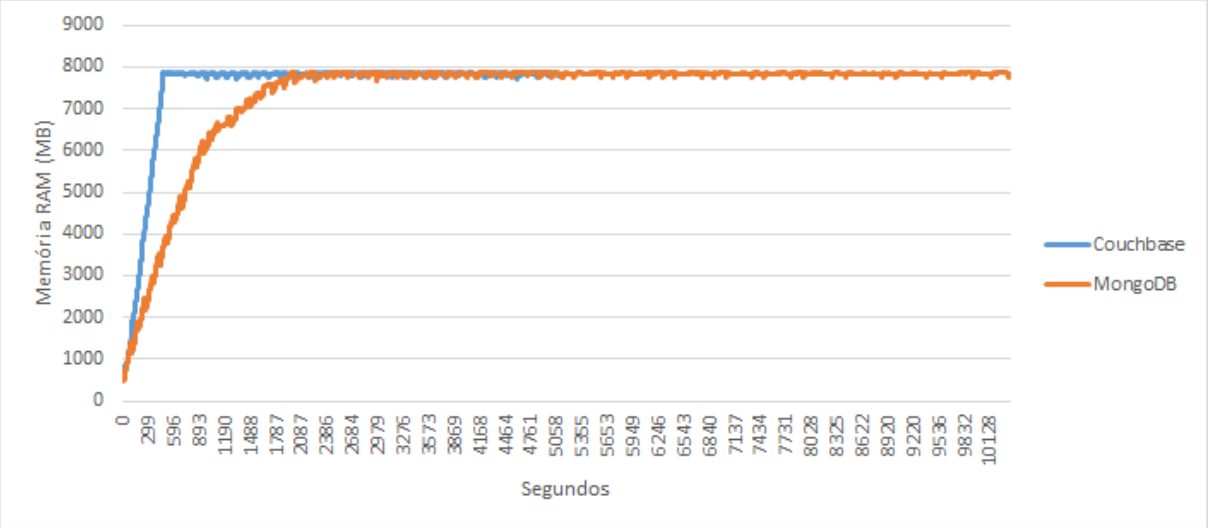
Quadro 53 - Uso de disco de Couchbase e MongoDB

	Uso da largura de banda do disco	Média de requisições de leitura/segundo	Média de requisições de escrita/segundo	Tamanho médio da requisição (em setores do disco)
Couchbase	83,92%	116,437	28,959	50,837
MongoDB	86,78%	18,728	24,457	164,767

O Quadro 53 detalha o uso de disco de ambos os bancos conforme medido pelo comando *iostat*. Pode-se notar que o percentual de largura de banda utilizada do dispositivo é bastante semelhante, assim como as requisições de escrita, porém o Couchbase faz requisições de leitura menores e em mais quantidade, enquanto que o MongoDB faz requisições maiores e em menor quantidade. Isto explica a melhor utilização do disco do MongoDB.

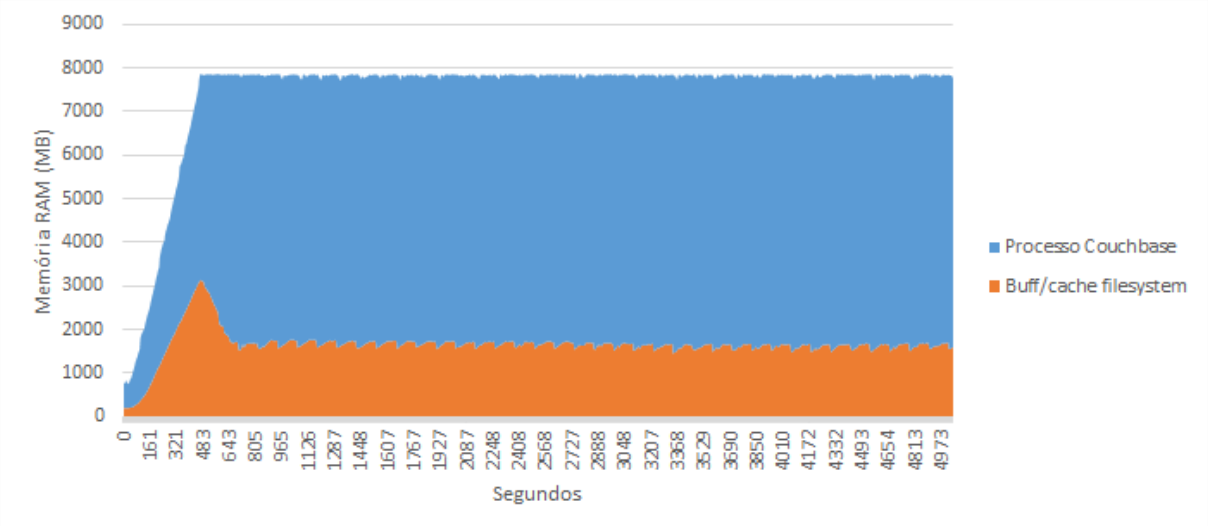
Outro recurso de *hardware* avaliado foi o consumo de memória RAM do servidor. O servidor possuía 8GB de memória e pode-se observar na Figura 31 as diferenças no consumo da memória RAM dos bancos em questão para a mesma carga de dados.

Figura 31 - Consumo de memória RAM de Couchbase e MongoDB



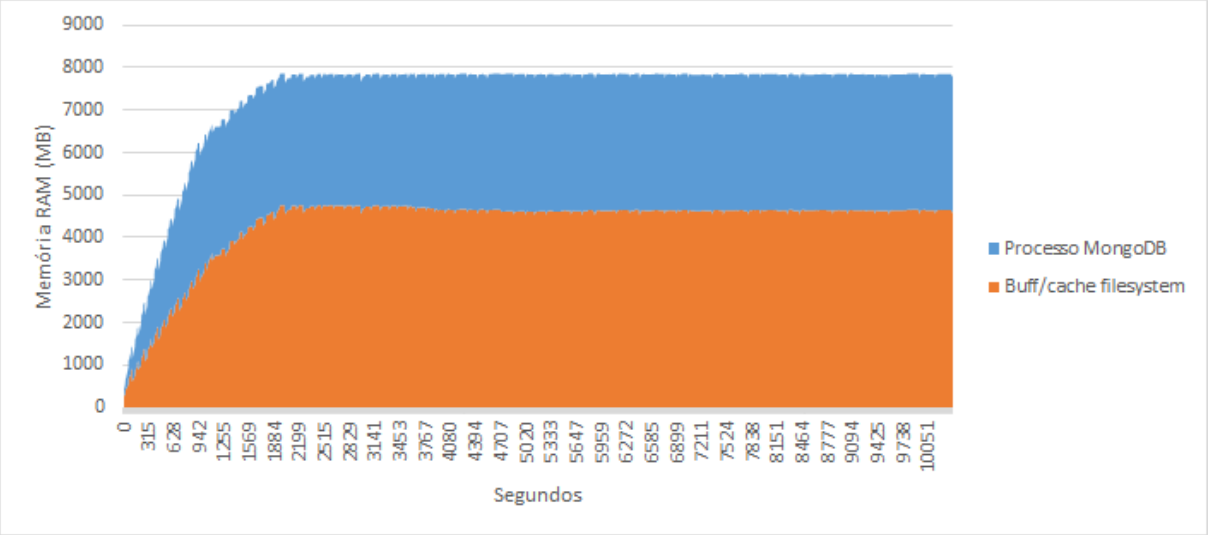
O Couchbase gerencia automaticamente a camada de cache e intercala com o armazenamento em disco, isso para garantir que haja espaço em memória suficiente para manter o desempenho do banco de dados. A memória alocada é dividida em três modos, um para armazenar índices, o *cache* de dados e serviço de consultas o qual armazena planos de execução para garantir maior velocidade nas operações (COUCHBASE, 2010). Na Figura 32 é possível observar a composição do consumo de memória do processo do banco e do buffer e cache do *filesystem*.

Figura 32 - Composição do consumo de memória do Couchbase



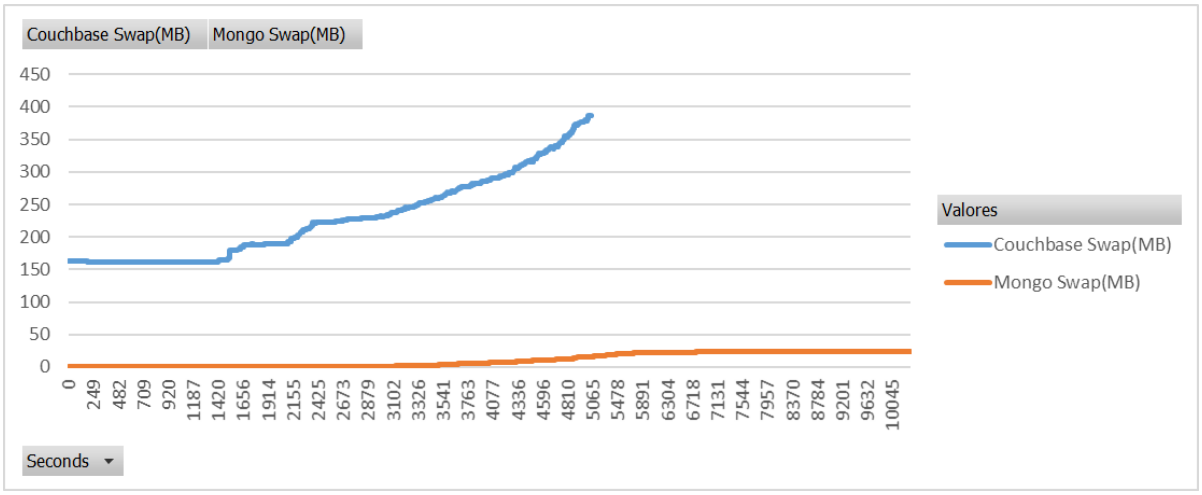
O MongoDB armazena os índices na memória para agilizar o acesso aos dados, além de utilizar dois caches: do *filesystem* e do *storage engine* (no caso, WiredTiger). O tamanho do cache do WiredTiger é configurável (nos testes foi mantido o padrão de 50% da memória menos 1GB), e todo o restante da memória disponível é utilizado pelo cache do *filesystem* (MONGODB, 2009).

Figura 33 - Composição do consumo de memória do MongoDB



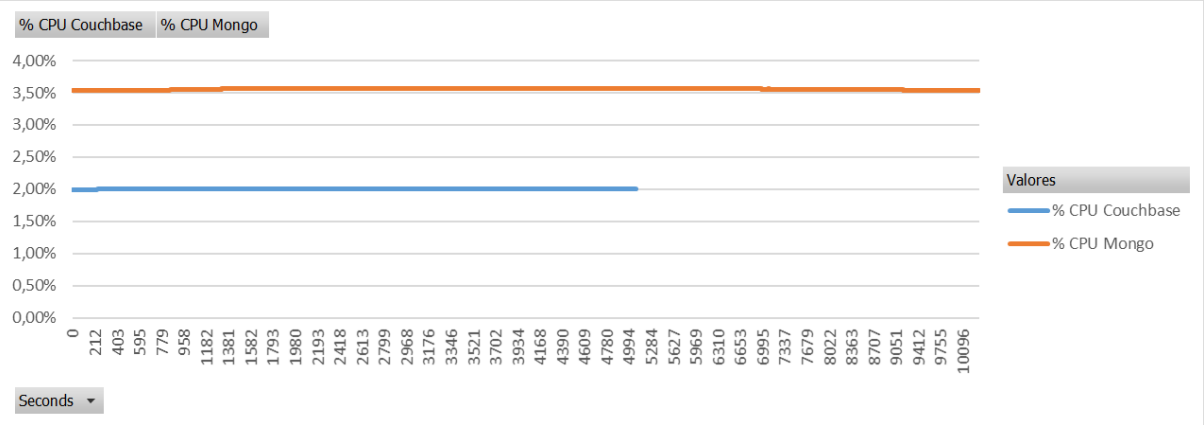
É possível notar uma clara diferença entre a composição do consumo de memória do Couchbase (Figura 32) e do MongoDB (Figura 33), pois o segundo utiliza muito mais o cache do sistema de arquivos do que o primeiro.

Figura 34 - Swap de Couchbase e MongoDB



Conforme pode ser observado na Figura 34, ambos bancos de dados realizaram operações de *swap*. Porém nota-se uma clara tendência crescente no uso desse recurso pelo Couchbase, que fez muito mais *swap* do que o MongoDB. A menor utilização de *swap* pelo segundo é explicada pela sua configuração padrão que limita o uso de memória de seu cache e gerencia a utilização de memória dentro deste limite.

Figura 35 - Consumo de CPU de Couchbase e MongoDB



Ambas chamadas partiram do cliente utilizando 4 threads, e a distribuição da utilização da CPU ficou a cargo do banco de dados, o resultado do consumo deste recurso consta na Figura 35.

A métrica de *througuput* também foi avaliada na iteração 15 e 16, a fim de observar o comportamento dos bancos durante a execução da carga. A Figura 36, Figura 37 e Figura 38 apresentam graficamente a evolução das operações por segundo de cada um dos bancos de dados.

Figura 36 - *Throughput* de MongoDB e Couchbase Seg 0 até 3240

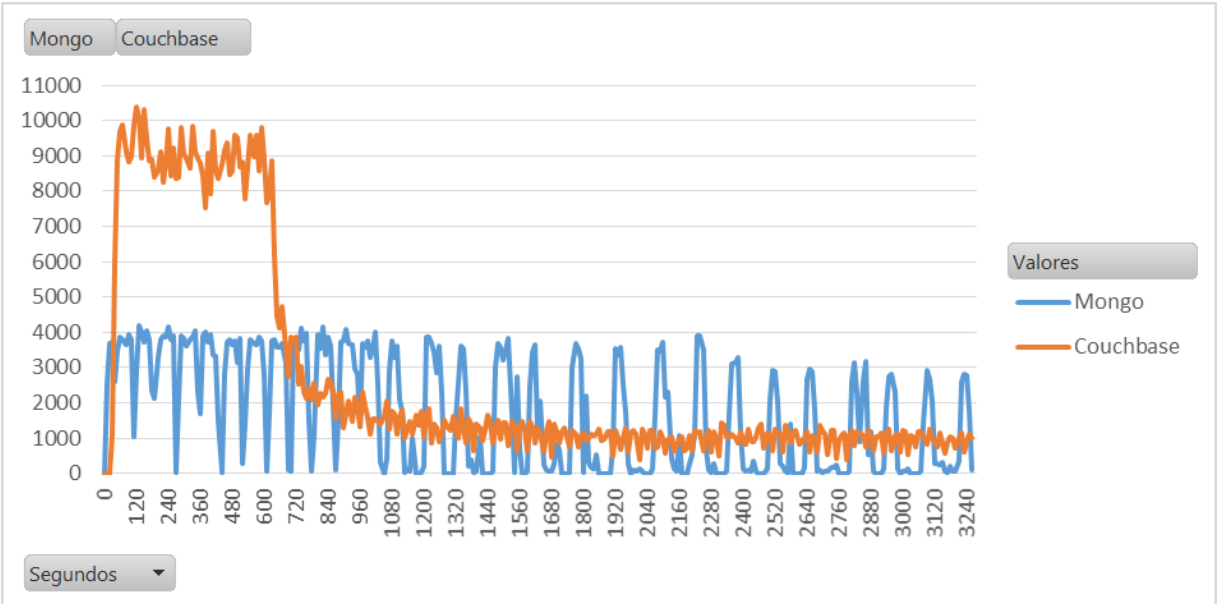


Figura 37 - *Throughput* de MongoDB e Couchbase Seg 3250 até 6500

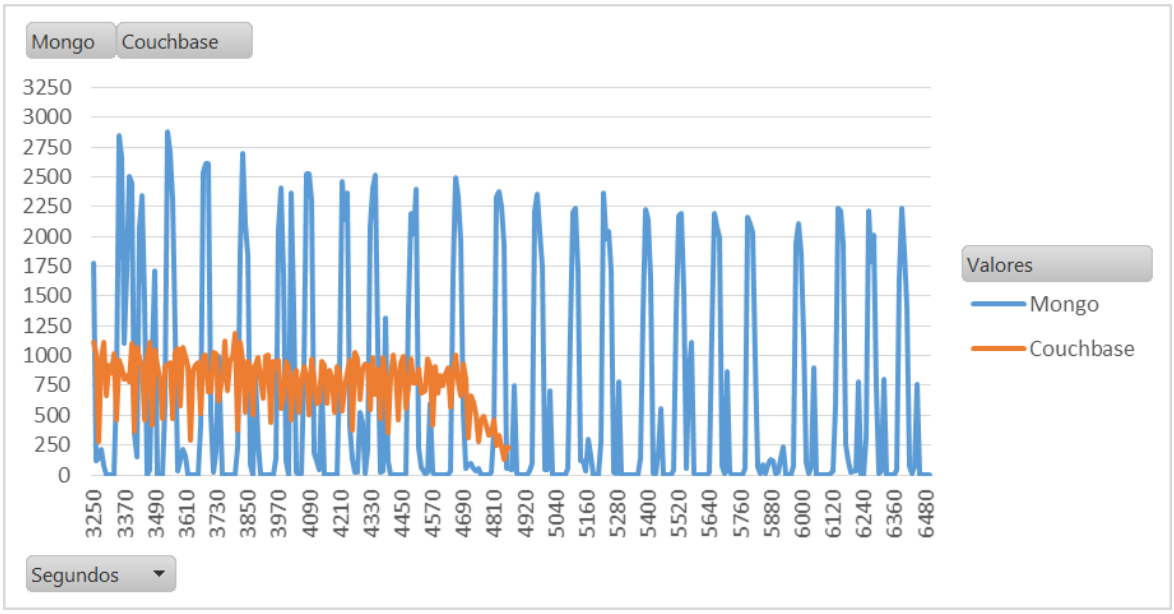
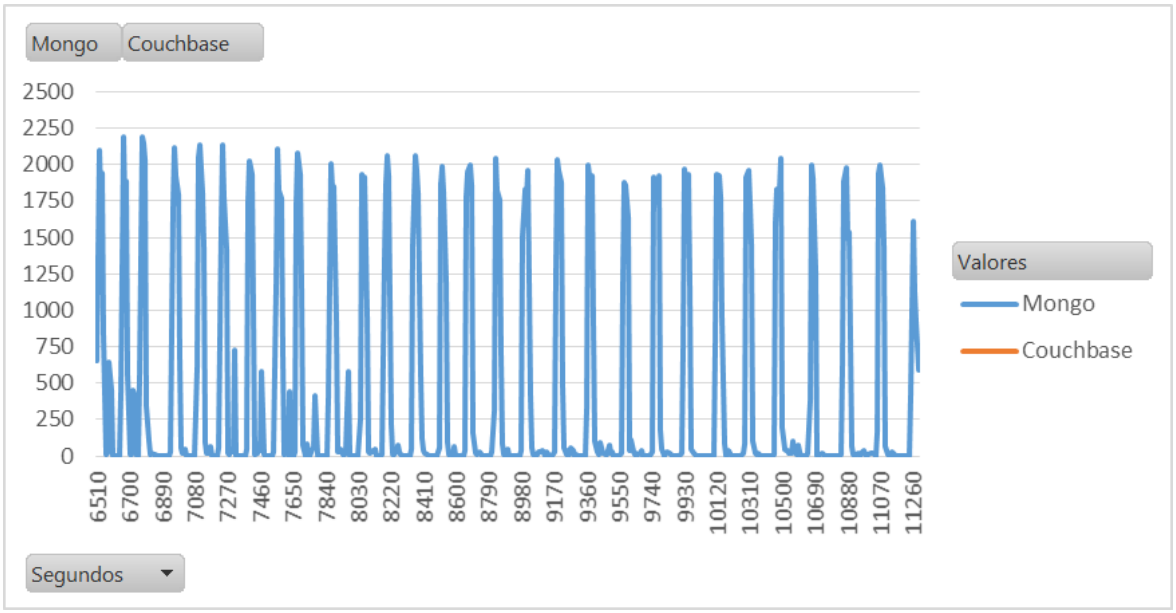


Figura 38 - *Throughput* de MongoDB e Couchbase Seg 6510 até 11260



Conforme já observado na análise estatística o Couchbase possui uma média de *throughput* maior que o MongoDB, porém se observa nas figuras acima que o MongoDB possui uma constância nesta medida, enquanto o Couchbase tem um queda brusca ainda no primeiro gráfico. Isso pode ser explicado pela dependência de memória RAM do Couchbase, já comentada nesta seção.

Assim como as demais medidas, a latência de leitura em microssegundos (us) durante a execução da iteração 15 do Mongo e 16 do Couchbase, está apresentada

na Figura 27. Essa informação se refere ao tempo de processamento da operação de leitura do cliente, processamento no servidor e retorno da informação para o cliente.

Figura 39 – Latência de leitura Couchbase e MongoDB segundo 0 até 2520

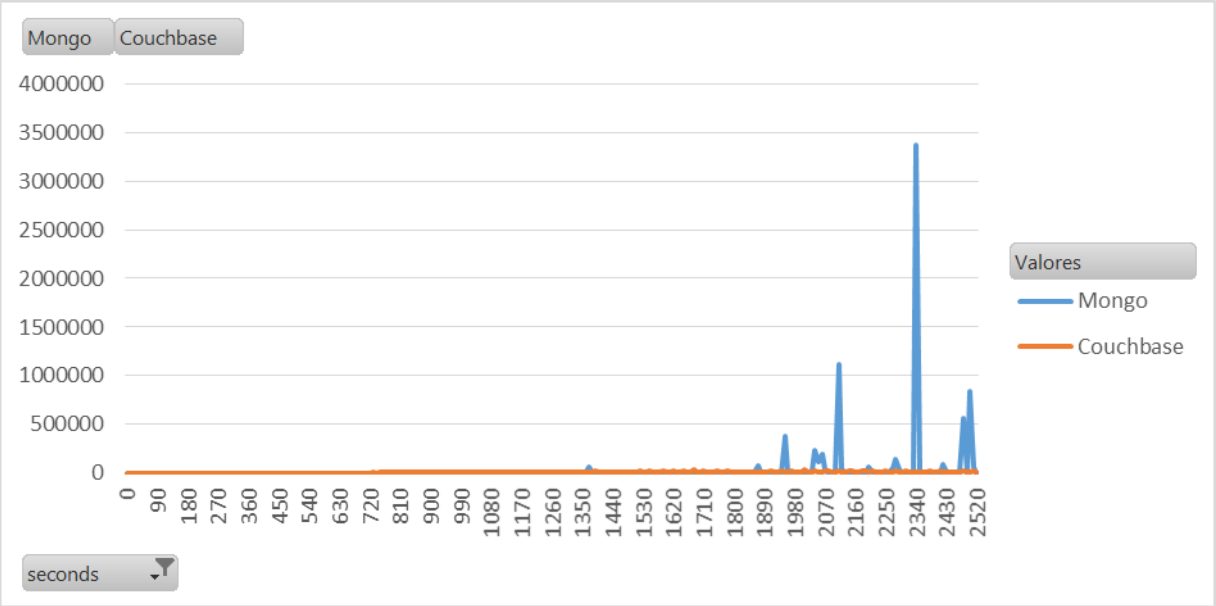


Figura 40 – Latência de leitura Couchbase e MongoDB segundo 2530 até 5040

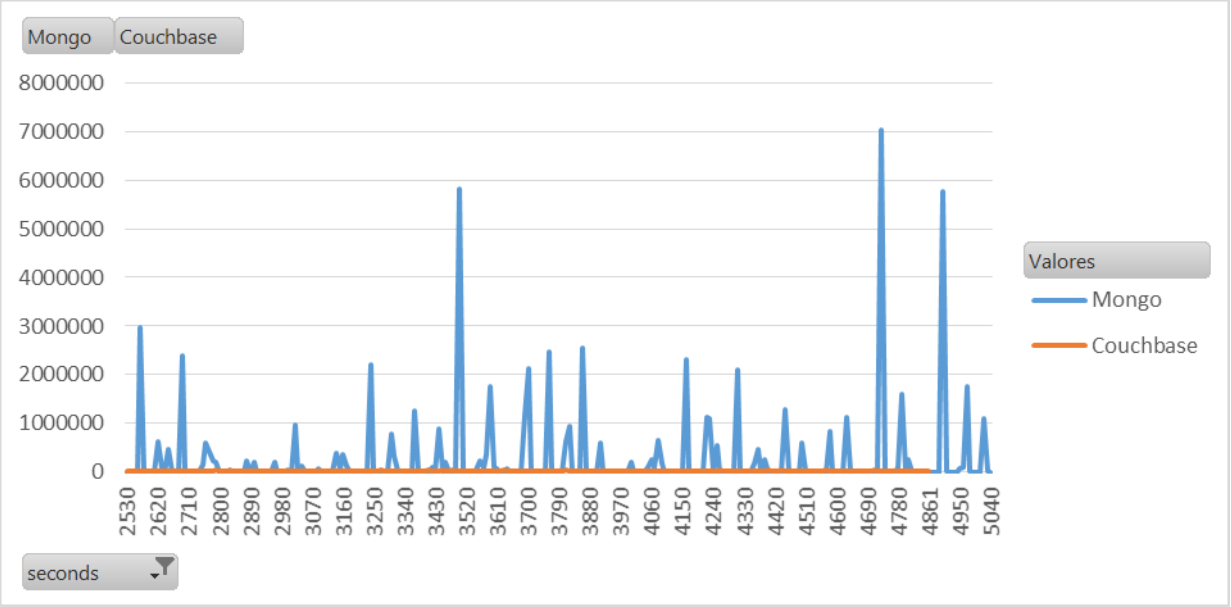


Figura 41 – Latência de leitura Couchbase e MongoDB segundo 5050 até 7480

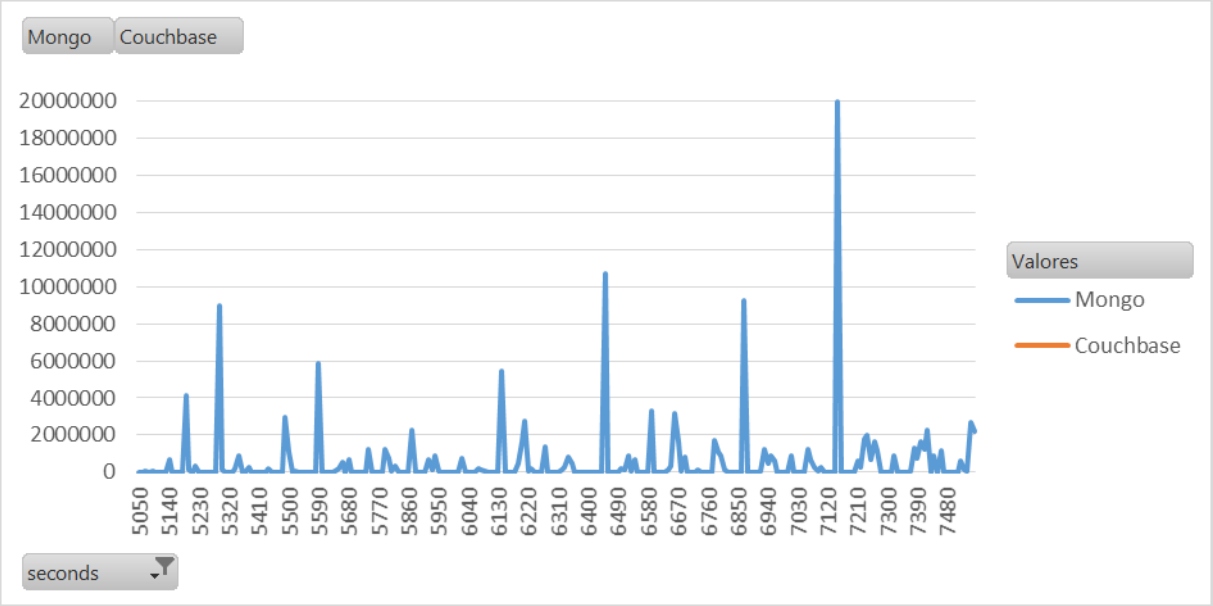
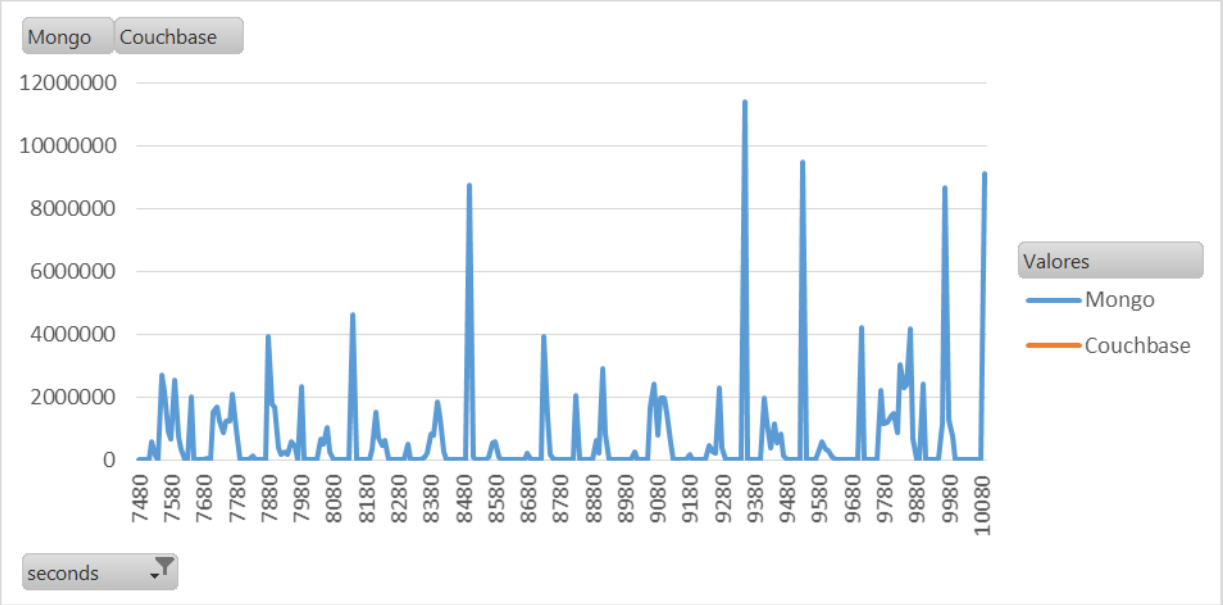


Figura 42 – Latência de leitura Couchbase e MongoDB segundo 7570 até 11082



De acordo com o resultado obtido na análise estatística, a média da latência de leitura do MongoDB é maior do que a do Couchbase, isso se reafirma se observado o gráfico apresentado. É possível observar que a latência nas operações de leitura do MongoDB é instável, sendo que o pico alcançado foi de 19,980212 segundos já o Couchbase foi 0,05955584 segundo.

Além disso durante a execução da carga, a latência do Couchbase se mantém consideravelmente abaixo do MongoDB, tornando a sua linha praticamente

imperceptível na comparação gráfica. Devido a isso, nas figuras abaixo está apresentado graficamente o resultado isolado do Couchbase.

Figura 43 - Latência de leitura Couchbase

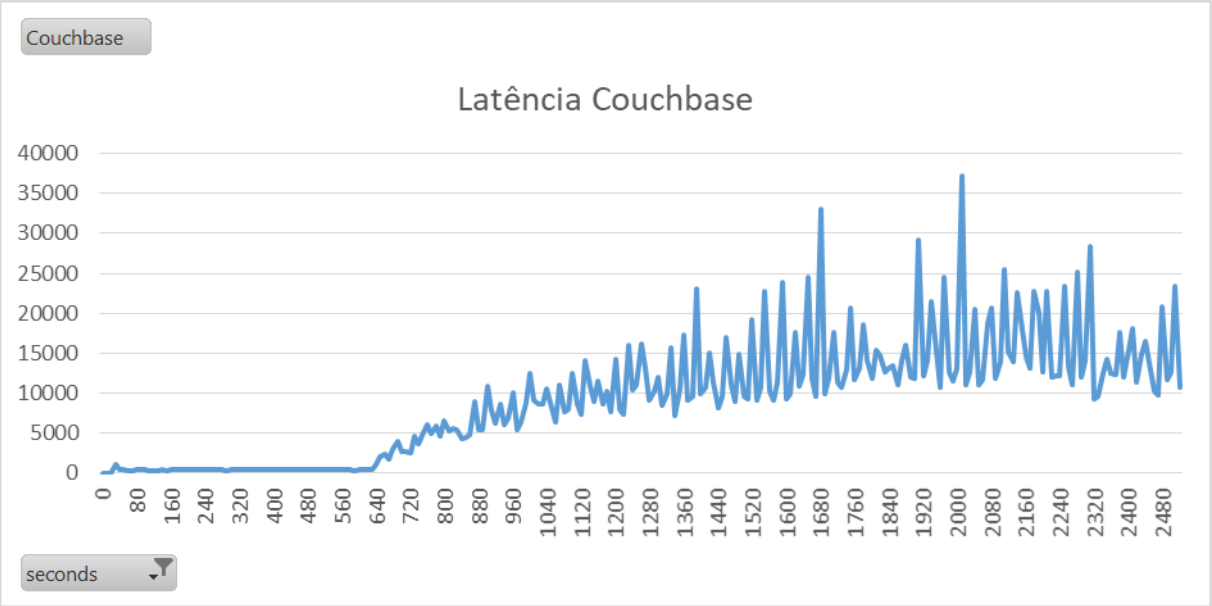
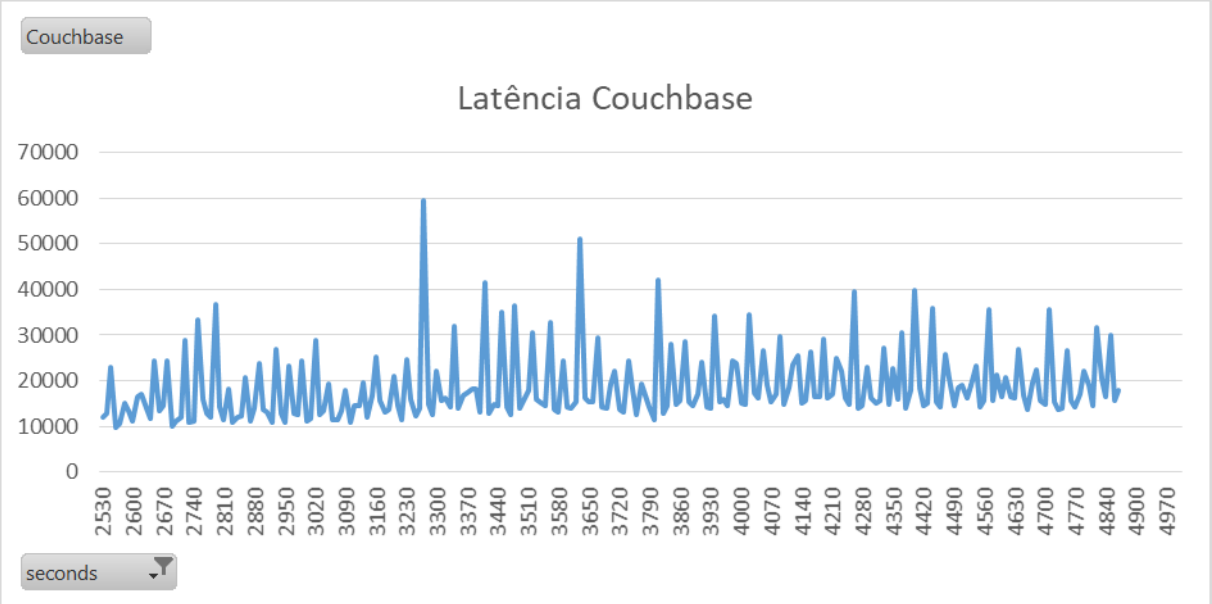


Figura 44 - Latência de leitura Couchbase



Do mesmo modo que é importante apresentar os resultados da latência de leitura, é necessário apresentar os resultados da latência de inserção, os quais podem ser visualizados na Figura 45. Essa informação se refere ao tempo de processamento da operação de inserção do cliente, processamento no servidor e retorno da informação para o cliente.

Figura 45 – Latência de inserção de Couchbase e Mongo 0 a 2500 segundos

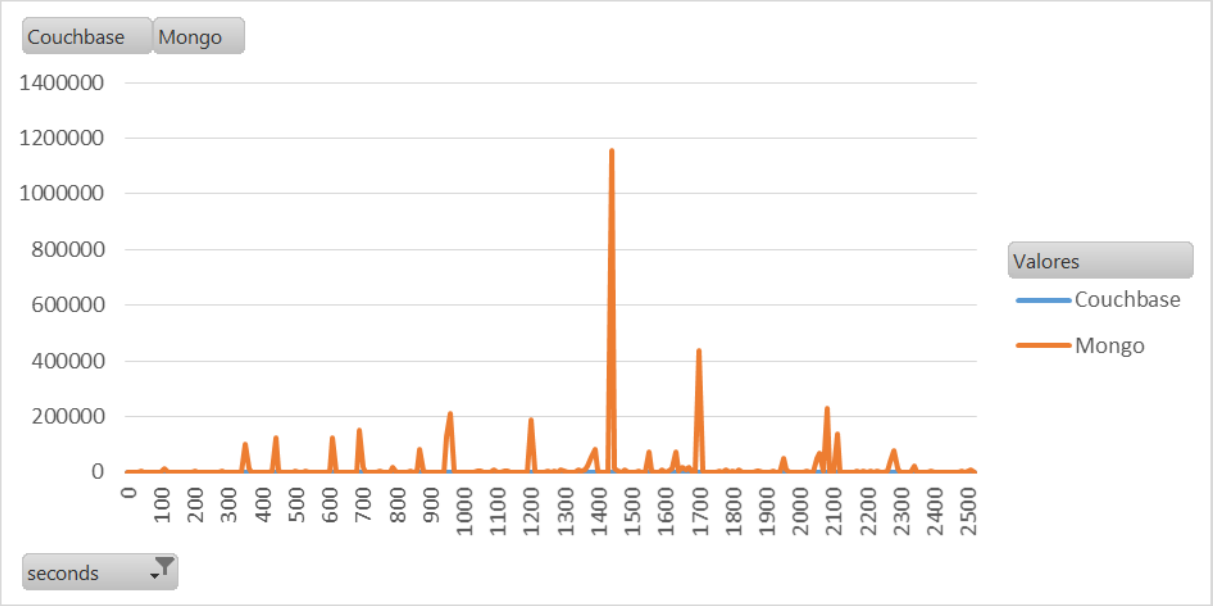


Figura 46 - Latência de inserção de Couchbase e Mongo 2510 a 5470 segundos

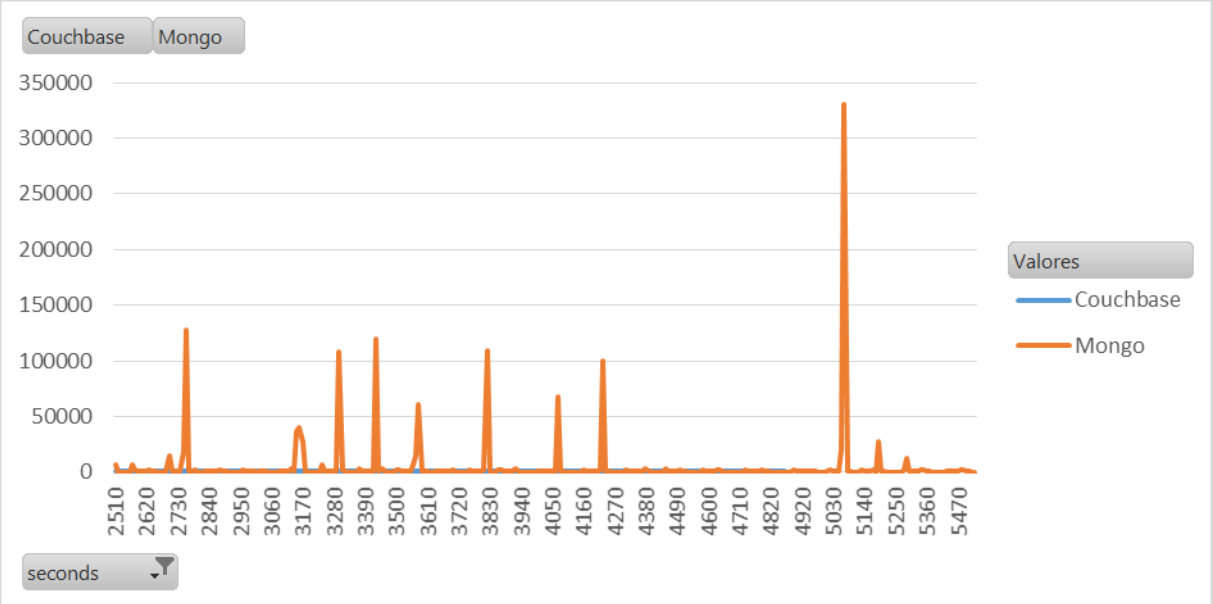


Figura 47 - Latência de inserção de Couchbase e Mongo 2510 a 5470 segundos

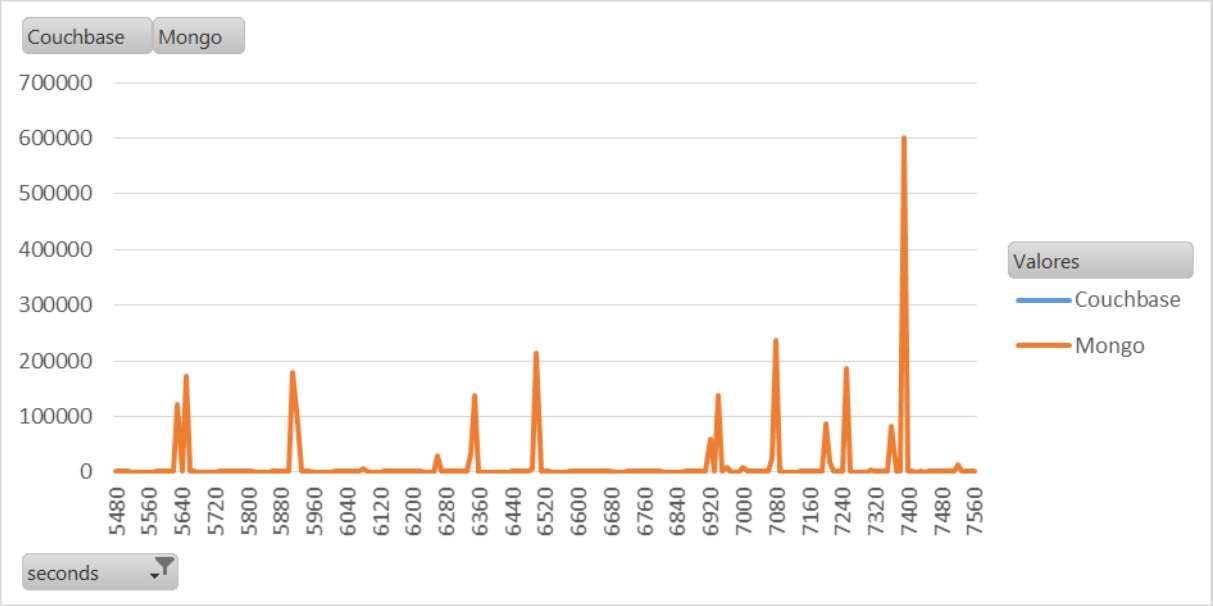
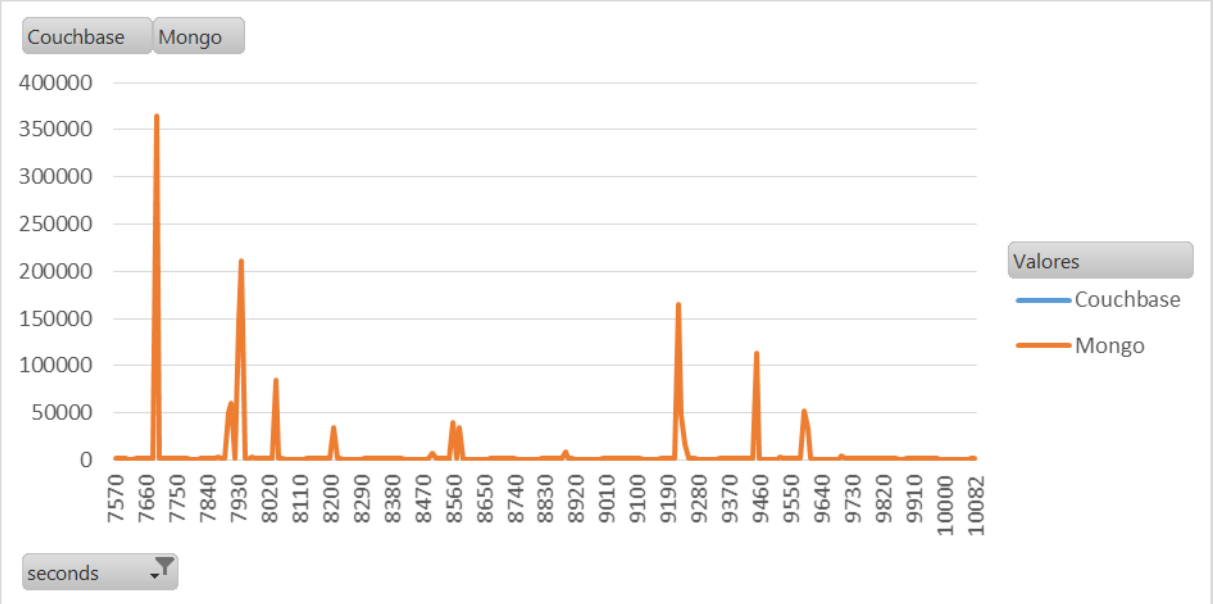


Figura 48 - Latência de inserção de Couchbase e Mongo 7570 a 11082 segundos



Novamente se observa, que o Mongo, possui a latência mais elevada do que o Couchbase, tornando sua linha graficamente indefinida. Sendo assim, gerou-se representações gráficas exclusivas para o Couchbase nas figuras abaixo.

Figura 49 - Latência de inserção Couchbase

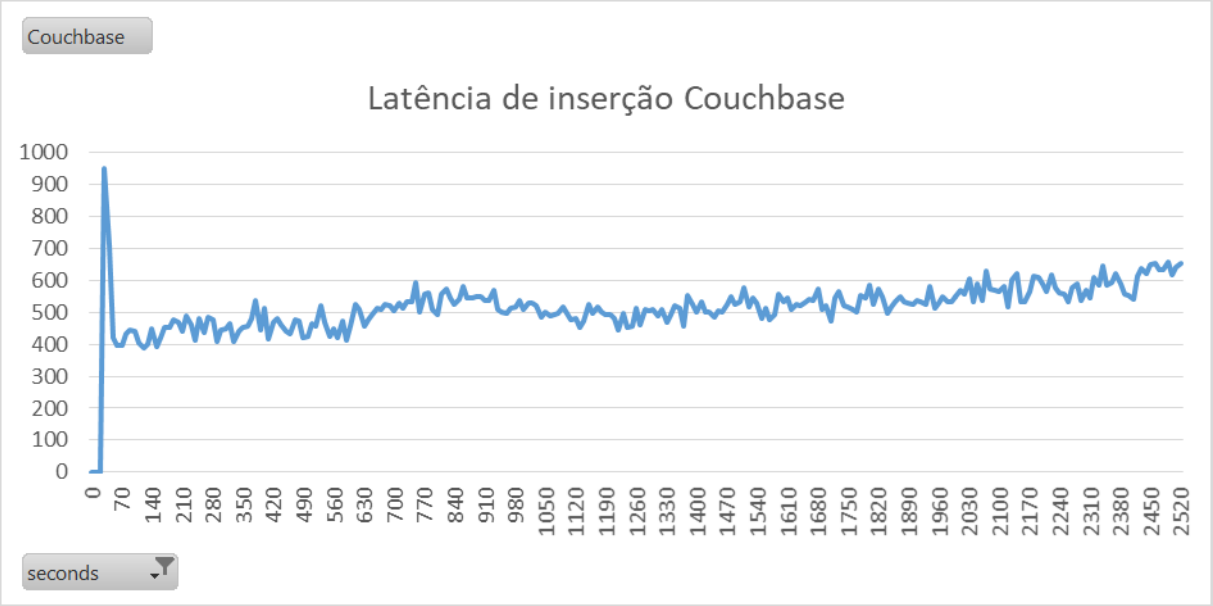
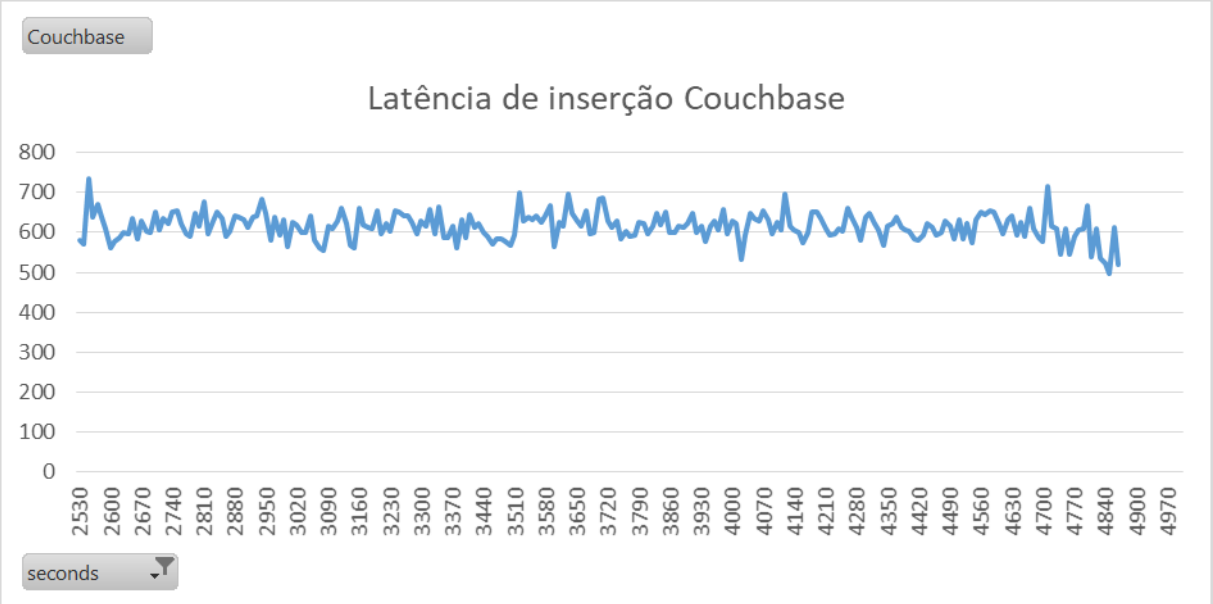


Figura 50 - Latência de inserção Couchbase



Com análise gráfica facilita a interpretação dos resultados apresentados na análise estatística, onde as médias de latência do Couchbase são menores do que do Mongo.

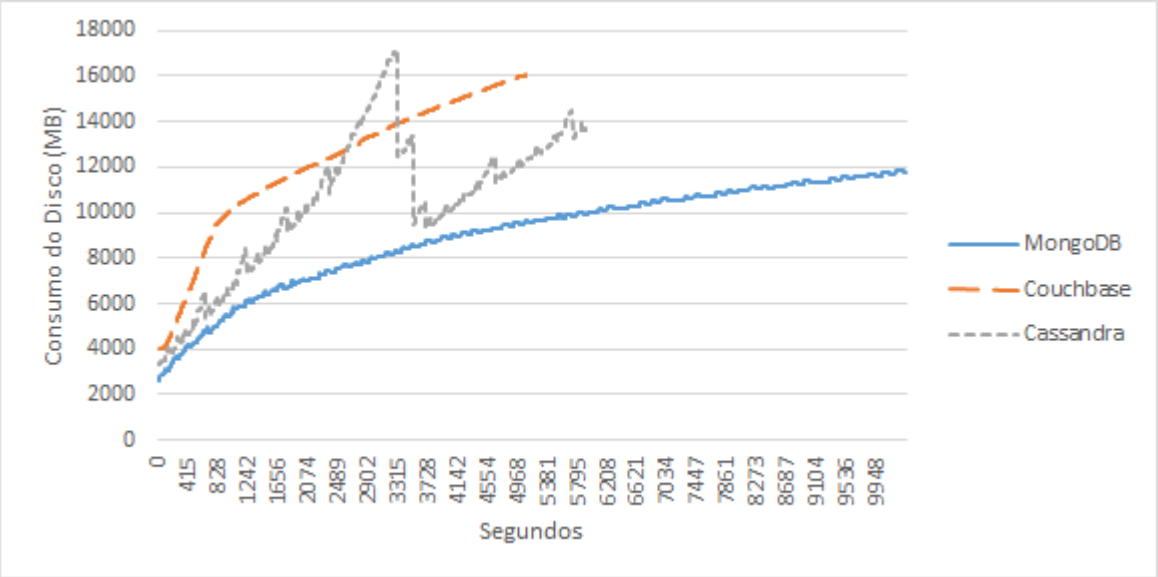
3.6.2.3 Cassandra e Bancos orientados a documentos

Conforme já citado anteriormente, a carga de dados executada para o Cassandra foi a mesma carga dos bancos orientados a documentos, com um tamanho

de 10GB, sendo que 75% foram operações de inserção e 25% foram operações de leitura de dados.

A Figura 51 apresenta a utilização do espaço em disco do Cassandra em relação aos bancos de dados orientados a documento MongoDB e Couchbase, já apresentados na seção 3.6.2.2, sendo que eixo X representa o tempo de execução do teste e o eixo Y o espaço em disco utilizado, observando que o total do espaço disponível é de 20GB.

Figura 51 - Consumo de disco de Cassandra, Couchbase e MongoDB



É possível notar que o Cassandra tem um consumo inicial maior de espaço em disco, que é reduzido drasticamente quando o processo de compactação automática (*minor compaction*) é executado, por volta dos 54 minutos e 30 segundos (3270 segundos) de teste.

Quadro 54 - Uso de disco de Cassandra, Couchbase e MongoDB

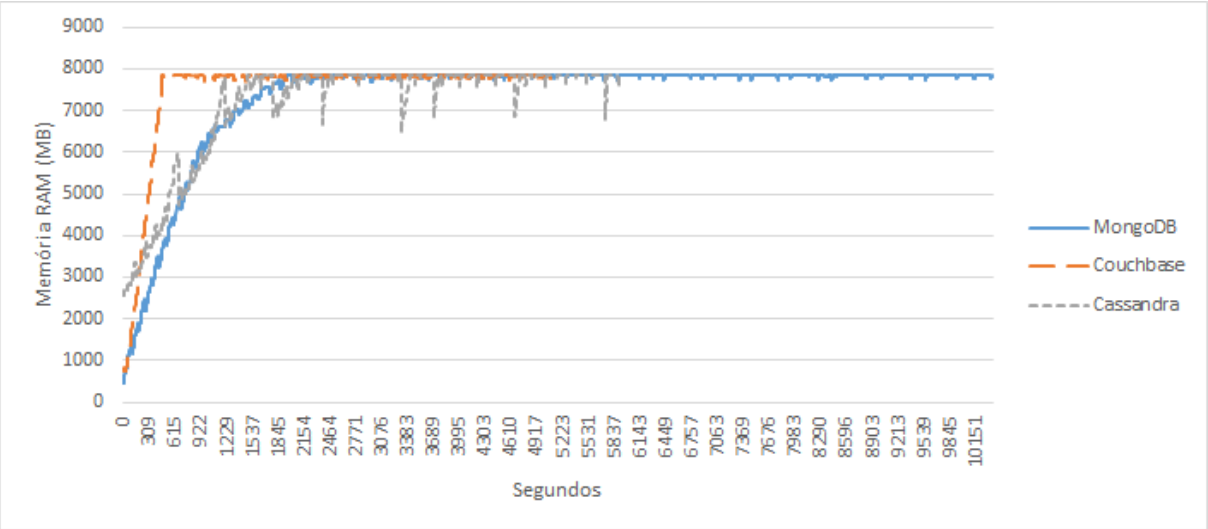
	Uso da largura de banda do disco	Média de requisições de leitura/segundo	Média de requisições de escrita/segundo	Tamanho médio da requisição (em setores do disco)
Cassandra	59,37%	146,280	9,974	143,053
Couchbase	83,92%	116,437	28,959	50,837
MongoDB	86,78%	18,728	24,457	164,767

O Quadro 54 detalha o uso de disco do Cassandra em relação aos bancos de documentos avaliados na seção 3.6.2.2. Pode-se notar que o Cassandra utiliza menos largura de banda do dispositivo do que os bancos de documentos, e faz mais requisições de leitura do que o Couchbase, porém muito menos requisições de escrita

do que ambos os bancos de documentos. A respeito do tamanho médio da requisição, no Cassandra ele é um pouco menor do que no Couchbase.

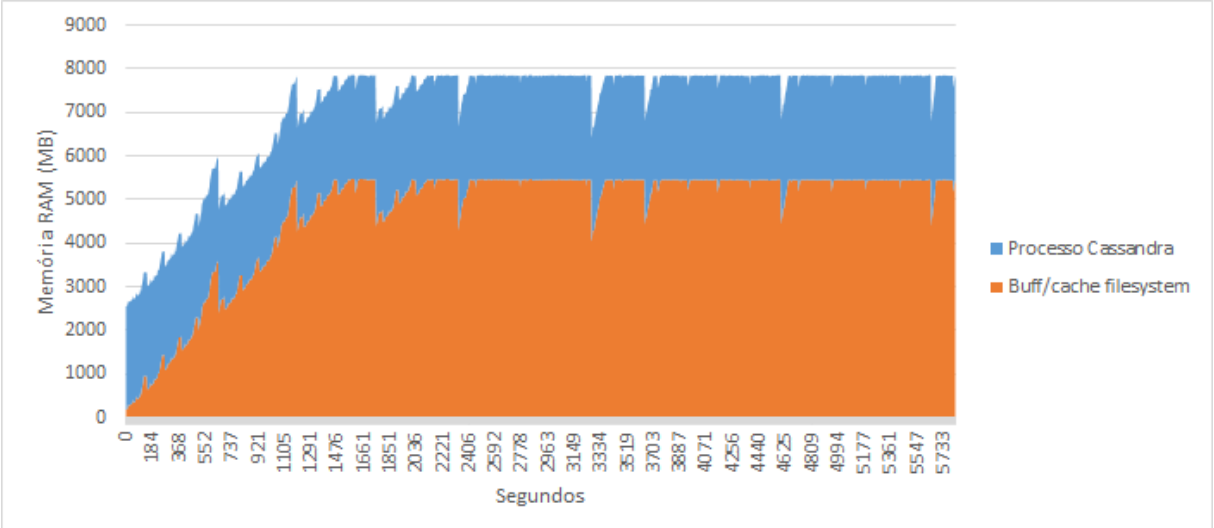
Quanto ao consumo de memória RAM, dos 8GB disponíveis no servidor é possível observar na Figura 52 que a utilização é bastante semelhante por todos os bancos, mas que no Cassandra há picos de alocação e liberação de memória que não ocorrem nem no Couchbase nem no MongoDB.

Figura 52 - Consumo de memória RAM de Cassandra, Couchbase e MongoDB



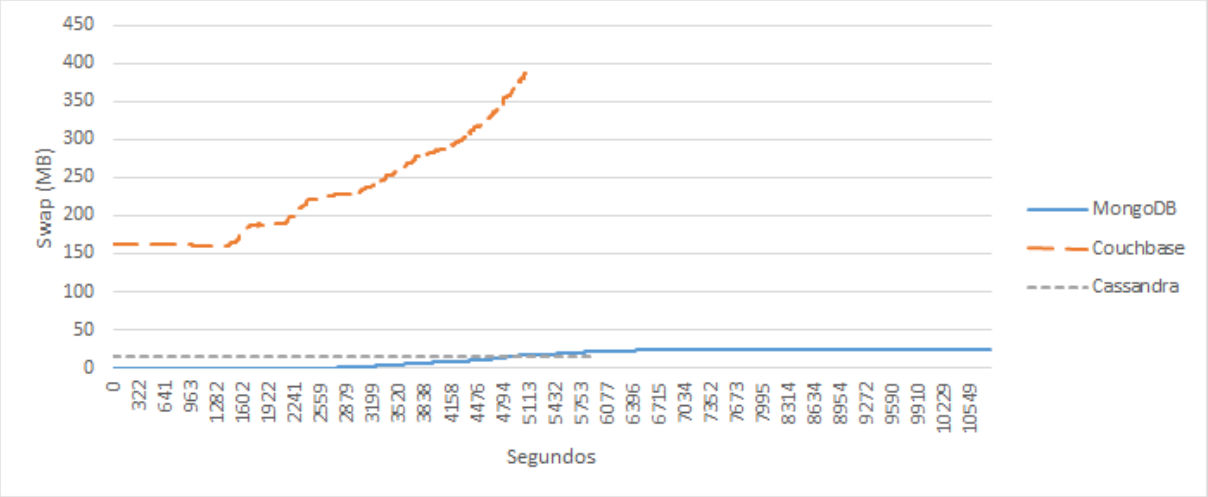
Quanto à composição do consumo de memória do Cassandra, que pode ser avaliada na Figura 53, é possível afirmar que o Cassandra em si não aloca muita memória, mas sim delega esta tarefa ao buffer e cache do sistema de arquivos. Aqui é possível observar que os picos da Figura 52 são originados no consumo do buffer e cache do *filesystem*, e não na alocação do próprio processo do banco de dados.

Figura 53 – Composição do consumo de memória do Cassandra



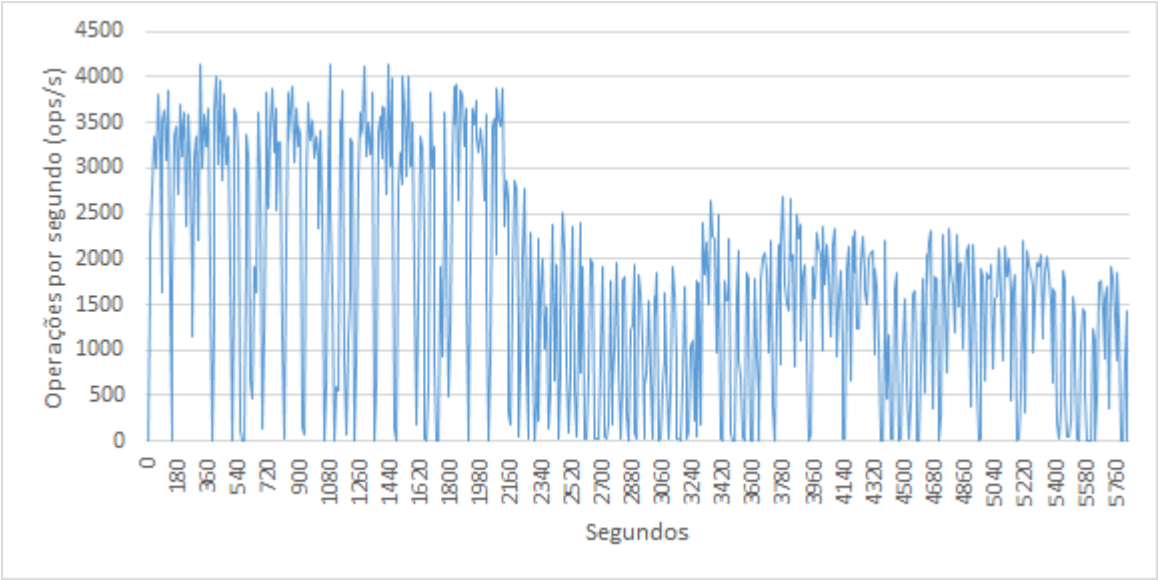
A respeito da composição do consumo de memória do Cassandra (Figura 53), é possível observar que ele é mais semelhante ao consumo do MongoDB (Figura 33) do que do Couchbase (Figura 32).

Figura 54 – *Swap* de Cassandra, Couchbase e MongoDB



A respeito do *swap*, é possível observar na Figura 54 que durante a sua execução o Cassandra não requereu nenhuma operação de *swap*, fato explicado pela pouca alteração na alocação de memória do próprio processo do banco de dados, uma vez que o sistema de arquivos não solicita *swap* para armazenar cache de arquivos.

Figura 55 – *Throughput* do Cassandra



Na Figura 55 é possível observar o *throughput* do Cassandra durante o teste. Pode-se notar que no início do teste há picos de 4.000 operações por segundo (ops/s) e a média se mantêm acima das 2.500 ops/s, porém em torno de 36 minutos de teste o *throughput* cai, ficando com uma média de 1.200 ops/s com picos de 2.500 ops/s.

Esta queda de desempenho é explicada pelo esgotamento da memória RAM disponível.

3.6.3 Dificuldades encontradas na fase de testes

Durante a execução dos testes foram encontradas algumas dificuldades em relação à utilização de alguns bancos.

Inicialmente, nos testes dos modelos chave-valor optou-se por testar Redis e Riak KV, porém durante a execução das cargas, apesar de estarem sendo executadas a limpeza e reinicialização do banco de dados, o tempo de execução aumentava exponencialmente. Foram avaliadas as configurações do banco conforme descrito na documentação oficial, porém não foi possível chegar a uma solução ao problema, sendo assim optou-se em substituí-lo pelo Aerospike para os testes estatísticos.

Em relação aos testes dos bancos *columnares*, não foi possível realizar as simulações com o banco de dado HBase e Accumulo, principalmente devido às suas dependências com Hadoop e Zookeper.

Inicialmente o objetivo era testar o Accumulo, porém mesmo com todos os componentes instalados no servidor o serviço do Zookeeper não iniciava em *background* e não era possível iniciar o banco de dados. Para contornar a situação, foi iniciado o serviço do Zoopeker em uma conexão via Putty, porém após alguns instantes rodando o serviço do banco, o mesmo caía sendo necessário iniciá-lo novamente de forma manual.

Sem sucesso na inicialização do Accumulo optou-se por avaliar o HBase, também pertencente à categoria de bancos de famílias de colunas. O processo de instalação deste banco ocorreu com sucesso, pois apesar de ele também possuir dependências do Hadoop e do Zookeper, estes serviços são distribuídos em conjunto com o banco de dados para execução em modo *standalone*, em que todos os serviços rodam na mesma JVM.

Porém ao configurar a carga do *benchmark* verificou-se que por padrão o YCSB é preparado para testar o HBase na mesma máquina, ou seja, ele deve ser executado *localhost*. Para contornar essa situação a comunidade orienta que se copiem os arquivos de configuração do banco de dados para o cliente.

Desta forma o banco se manteve *online* para início da carga, porém após executar parte das operações de inserção o serviço do banco de dados perdia a conexão com o Zookeeper e se encerrava automaticamente. Paralelamente o Zookeeper identifica que determinado nó do *cluster* falhou e tenta realocar os dados para outro nó, porém este segundo nó não existe e o processo de redirecionamento volta a tentar comunicação com o mesmo servidor, que já está encerrado.

Em busca da solução avaliaram-se possíveis variáveis que poderiam causar travamento da JVM e causar esse efeito. Foi avaliado se ocorram operações de *swap*, se haviam se esgotado o número de *open file handles* e foi configurado no Zookeper um *timeout* de sessão alto e para não haver limitações na quantidade de clientes conectados. Por fim, também foram avaliados os tempos de pausa na JVM para execução do *garbage collector* (GC) do Java, porém todas as variáveis analisadas estavam de acordo, não justificando o comportamento apresentado.

Nos testes dos bancos orientados a documentos o OrientDB também foi avaliado inicialmente, porém não foi possível confirmar que as configurações de persistência definidas estavam sendo carregadas e utilizadas pelo banco de dados (problema já reportado há dois anos no *issue* de número 4181 do repositório oficial), minando a confiabilidade nos resultados.

Outro fator decisivo para a decisão de avaliar outra tecnologia ao invés do OrientDB foi o parecer do relatório da Gartner de 2015 de que o banco atingiu uma das menores pontuações em suporte e documentação, além de que muitos clientes encontram *bugs* no mesmo (GARTNER, 2015).

CONCLUSÃO

Com os resultados obtidos e apresentados no Capítulo 3 buscou-se responder a lacuna encontrada no início do projeto, que consiste em identificar quais dos bancos de dados NoSQL estudados oferecem melhor desempenho no quesito velocidade no processamento de dados.

A presente pesquisa possui como objetivo avaliar e analisar o desempenho de tecnologias de bancos de dados NoSQL a fim de solucionar o problema encontrado, bem como disponibilizar à comunidade uma ferramenta para auxiliar a tomada de decisão. Nos Capítulos 2 e 3 estão apresentados o *survey* e os resultados dos testes estatísticos executados para que os objetivos fossem atingidos. Além dos resultados apresentados nesse documento, vale ressaltar que o objetivo de publicação de artigo foi atingido e seu resultado consta publicado conforme referência Rockenbach, Anderle, *et al.* (2017).

Foram levantadas quatro hipóteses; A primeira delas afirma que o desempenho dos bancos de dados chave-valor é estatisticamente diferente entre as tecnologias avaliadas, essa hipótese foi corroborada com base na análise de significância estatística das amostras.

Os bancos avaliados da categoria chave-valor foram Redis e Aerospike. Nas métricas avaliadas (*throughput, runtime*, latência de leitura e latência de gravação) observou-se que o Aerospike possui um desempenho melhor que o Redis para o cenário onde os testes foram aplicados, considerando operações de leitura e escrita, sem persistência em disco habilitada. Além disso, observou-se que na utilização de recursos do servidor o Aerospike exigiu menos recursos de memória do que o Redis, porém com um consumo maior de recursos de CPU.

A segunda hipótese descrita não pôde ser avaliada, uma vez que em função do cronograma da presente pesquisa o desempenho dos bancos de dados família de colunas não foi avaliado.

Já a terceira hipótese, que afirma que o desempenho dos bancos de dados orientados a documentos é estatisticamente diferente entre as tecnologias avaliadas, foi corroborada, pois conforme apresentado nos resultados, as tecnologias testadas não possuem significância.

Os bancos avaliados da categoria orientados a documentos foram MongoDB e Couchbase. Nas métricas avaliadas (*throughput, runtime*, latência de leitura e latência de gravação) observou-se que o Couchbase possui um desempenho melhor que o MongoDB, para o cenário onde os testes foram aplicados, considerando 25% de operações de leitura e 75% de operações de escrita.

Em relação ao consumo dos recursos do servidor, o consumo de memória RAM foi equivalente, porém o Couchbase executou mais *swap* do que o MongoDB. Já no uso de disco observou-se que o MongoDB gerencia melhor esse recurso em função de uma funcionalidade de compactação, e referente ao consumo de CPU é possível afirmar que o Couchbase distribui melhor a carga para os núcleos, pois sua média de utilização foi menor.

A quarta hipótese afirma que o desempenho dos bancos de dados de grafos é estatisticamente diferente entre as tecnologias avaliadas e também não pôde ser testada, pois o *benchmark* selecionado em uma análise prévia contemplava os bancos deste modelo, porém como alguns se tratam de bancos de dados híbridos, no momento da aplicação dos testes verificou-se que o YCSB não atendia a eles no modo grafo ou sua implementação não estava homologada, sendo assim estas tecnologias não foram submetidas aos testes.

Este estudo foi aplicado nos cenários apresentados no item 3.4, sendo apenas uma ferramenta norteadora. Por se tratar de experimentos executados em cenários isolados, os resultados aqui expostos não são verdade absoluta para todas as realidades. Então, antes de optar por alguma ferramenta, os autores recomendam que sejam executados testes baseados nos cenários aos quais pretende-se aplicar determinada tecnologia de bancos.

Com base na recomendação apresentada, nota-se a importância da utilização de um método científico bem definido, não apenas para academia, mas para as empresas e a sociedade em geral. Pois, é através deste método que será possível expandir os resultados deste trabalho para outros cenários de teste e garantir que os resultados atingidos sejam efetivamente aplicados. O método garante que não há qualquer influência ou privilégio no processo de experimentação das ferramentas, garantido a confiabilidade e robustez da informação.

O trabalho proporcionou o conhecimento de uma área totalmente nova para os acadêmicos e possibilitou o contato com a pesquisa científica. Permitiu ampliar o entendimento de fundamentos de bancos de dados, estatística, testes com *benchmark*, além de apresentar e possibilitar o contato com uma quantidade considerável de tecnologias desta emergente área de bancos de dados NoSQL.

Além dos resultados já alcançados, pretende-se ainda realizar trabalhos futuros, tais como refinamento da pesquisa. Além de executar novos testes aplicando diferentes conjuntos de configurações, buscando otimizar o desempenho dos bancos de dados, para avaliar o comportamento dos bancos em cenários distintos. Ainda fica a opção de incluir outros bancos, ainda não contemplados nessa fase de testes.

Existe também a oportunidade de encontrar um *benchmark* que atenda aos bancos de grafos, pois estes possuem um mercado em ascensão destinado a aplicações complexas, as quais saem do nicho comercial e abrangem mercados governamentais e acadêmicos, tais como pesquisas meteorológicas e de georeferenciamento.

Esse conjunto de propostas também abrange novos nichos de pesquisa e estruturação de conhecimentos para a comunidade científica, primariamente através da publicação de novos artigos.

REFERÊNCIAS

ABADI, D. Consistency Tradeoffs in Modern Distributed Database System Design: CAP is Only Part of the Story. **Computer**, 45, n. 2, fev. 2012. 37-42.

ABRAMOVA, V.; BERNARDINO, J.; FURTADO, P. Which NoSQL Database? A Performance Overview. **Open Journal Databases (OJDB)**, 1, n. 2, 2014. 17-24. Disponivel em: <https://www.ronpub.com/OJDB-v1i2n02_Abramova.pdf>. Acesso em: 12 dez. 2016.

AEROSPIKE. Aerospike | High Performance NoSQL Database. **Aerospike**, 2012. Disponivel em: <http://www.aerospike.com/>.

AKGUL, F. **ZeroMQ**. Birmingham: Packt Publishing, 2013. ISBN 978-1-78216-104-2. Disponivel em: <https://books.google.com.br/books?id=u5N_A2_xwjcC&printsec=frontcover&hl=pt-BR>. Acesso em: 18 dez. 2016.

APACHE. Welcome to Apache Hadoop! **Apache Hadoop**, 2011. Disponivel em: <http://hadoop.apache.org/>. Acesso em: 03 Mar 2017.

APACHE ACCUMULO. Accumulo. **Accumulo**, 2008. Disponivel em: <https://accumulo.apache.org/>. Acesso em: 14 Mar 2017.

APACHE ACTIVEMQ. Apache ActiveMQ ™. **Apache ActiveMQ**. Disponivel em: <http://activemq.apache.org/>. Acesso em: 15 nov. 2016.

APACHE KAFKA. Apache Kafka. **Apache Kafka**. Disponivel em: <https://kafka.apache.org/>. Acesso em: 15 nov. 2016.

APACHE SOFTWARE FOUNDATION. Apache HBase. **Apache Software Foundation**, 2008. Disponivel em: <hbase.apache.org>. Acesso em: 21 Fev 2017.

APACHE SOFTWARE FUNDATION. Apache Cassandra. **Apache Cassandra**, 2008. Disponivel em: <http://cassandra.apache.org/>. Acesso em: 21 Fev 2017.

APACHE TINKERPOP. Apache TinkerPop. **Apache TinkerPop**, 2008. Disponivel em: <http://tinkerpop.apache.org/>. Acesso em: 12 mar. 2017.

BARU, C. et al. **Discussion of BigBench:** a proposed industry standard performance benchmark for big data. Technology Conference on Performance Evaluation and Benchmarking. [S.l.]: Springer. 2014. p. 44-63.

BASHO. Riak KV. **Riak KV**, 2009. Disponivel em: <http://basho.com/products/riak-kv/>.

BISQUERRA, R.; SARRIERA, J. C.; MATÍNEZ, F. **Introdução a Estatística:** Enfoque informático com o pacote estatístico SPSS. Porto Alegre: Artmed, 2007. ISBN 978-85-363-1136-4.

BREWER, E. Towards Robust Distributed Systems. **Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing**, New York, p. 7, Jul 2000. ISSN 1-58113-183-6.

CAO, W. et al. **Evaluation and Analysis of In-Memory Key-Value Systems**. 2016 IEEE International Congress on Big Data (BigData Congress). San Francisco: IEEE. 2016. p. 26-33.

CARLSON, J. L. **Redis in Action**. New York: Manning Publications Co., 2013.

CHANG, F. et al. Bigtable:A distributed storage system for structured data. **ACM Trans. on Computer Systems (TOCS)**, v. 26, p. 4, 2008.

CHUERI, L. D. O. V.; XAVIER, C. M. D. S. **Metodologia de Gerenciamento de Projetos no Terceiro Setor:** uma estratégia para a condução de projetos. Rio de Janeiro: Brasport, 2008. ISBN 9788574523590.

COOPER, B. F. et al. Benchmarking cloud serving systems with YCSB. **SoCC '10 Proceedings of the 1st ACM symposium on Cloud computing**, Indianapolis, 10 jun. 2010. 143-154. Disponivel em: <https://s.yimg.com/ge/labs/v1/files/ycsb-v4.pdf>. Acesso em: 17 dez. 2016.

COPELAND, R. **MongoDB Applied Design Patterns**. Sebastopol: O'Reilly, 2013.

COUCHBASE. NoSQL Database | Couchbase. **Couchbase**, 2010. Disponivel em: <https://www.couchbase.com/>. Acesso em: 04 mar. 2017.

COUCHDB. Apache CouchDB. **Apache CouchDB**, 2005. Disponivel em: <http://couchdb.apache.org/>. Acesso em: 20 Fev 2017.

CRAWFORD, S.; GOLDSMITH, S. **The Responsive City:** Engaging Communities Through Data-Smart Governance. São Francisco: John Wiley & Sons, 2014. 208 p. ISBN 978-1-118-91090-0.

CROCKFORD, D. The application/json Media Type for JavaScript Object Notation (JSON). **RFC 4627**, jul. 2006. Disponivel em: <http://www.rfc-editor.org/info/rfc4627>. Acesso em: 26 ago. 2015.

DANCEY, C. P. **Estatítica sem matemática para psicologia**. 3. ed. Porto Alegre: Artmed, 2006. ISBN 978-85-363-0688-9.

DAS, V. **Learning Redis**. Birmingham: Packt Publishing, 2015. ISBN 978-1-78398-012-3.

DB-ENGINES. DB-Engines - Knowledge Base of Relational and NoSQL Database Management Systems. **DB-Engines**. Disponivel em: <http://db-engines.com/en/>. Acesso em: 15 nov. 2016.

DEAN, J.; GHEMAWAT, S. MapReduce: simplified data processing on large clusters. **Communications of the ACM**, 51, 2008. 107-113.

DECANDIA, G. et al. Dynamo: amazon's highly available key-value store. **ACM SIGOPS operating systems review**, Stevenson, v. 41, n. 6, p. 205,220, Outubro 2007.

DEKA, G. C. A survey of cloud database systems. **IT Professional**, v. 16, n. 2, p. 50-57, abr. 2014.

DORMANDO. Redis VS Memcached (slightly better bench). **LiveJournal**, 21 set. 2010. Disponivel em: <http://dormando.livejournal.com/525147.html>. Acesso em: 16 nov. 2016.

FACEBOOK. Facebook Graph Benchmark. **GitHub**, 2015. Disponivel em: <https://github.com/facebookarchive/linkbench>. Acesso em: 31 Mar 2017.

FARBER, L. **Estatística aplicada**. 4. ed. São Paulo: Pearson Prentice Hall, 2010. ISBN 978-85-7605-372-9.

FÁVERO, L.; FÁVERO,. **Estatística Aplicada:** Para Cursos de Administração, Contabilidade e Economia com Excel e SPSS. Rio de Janeiro: Elsevier Brasil, 2015. 480 p. ISBN 978-85-352-6356-5.

FERREIRA, D. F. **Estatística Básica**. Lavras: UFLA, 2005. 664 p. ISBN 85-87692-23-2.

FIELD, A. **Descobrindo a estatística e usando o SPSS**. Tradução de Lorí Viali. 2. ed. Porto Alegre: Artmed, 2009. ISBN 978-85-363-2018-2.

FIELD, A. **DISCOVERING STATISTICS:** USING SpSS. 3. ed. Dubai: Sage, 2009. ISBN 978-1-84787-906-6.

FOWLER, ; SADALAGE, P. J. **NoSQL Essencial:** Um Guia Conciso para o Mundo Emergente da Persistência Poliglota. São Paulo: Novatec, 2013. 216 p. ISBN 9788575223383.

FOWLER, A. **NoSQL For Dummies**. New Jersey: John Wiley & Sons, 2015. ISBN 978-1-118-90578-4. Disponivel em: <https://books.google.com.br/books?id=O_UwBgAAQBAJ&printsec=frontcover&hl=pt-BR>. Acesso em: 12 nov. 2016.

GALBRAITH, P. **Developing Web Applications with Apache, MySQL, memcached, and Perl**. Indianapolis: Wiley Publishing, 2009. ISBN 978-0-470-41464-4. Disponivel em:

<https://books.google.com.br/books?id=daDAnXPnRkcC&printsec=frontcover&hl=pt-BR>. Acesso em: 13 nov. 2016.

GAMMA, E. et al. **Padrões de projeto:** soluções reutilizáveis de software orientado a objetos. São Paulo: Bookman, 2000. Disponivel em: <https://books.google.com.br/books?id=U91CYCqTCgkC&printsec=frontcover&hl=pt-BR>. Acesso em: 04 nov. 2016.

GARG, N. **Learning Apache Kafka:** Second Edition. 2ª. ed. Birmingham: Packt Publishing, 2015. ISBN 978-1-78439-309-0. Disponivel em: <https://books.google.com.br/books?id=mi_WBgAAQBAJ&printsec=frontcover&hl=pt-BR>. Acesso em: 15 nov. 2016.

GARTNER. **Magic Quadrant for Operational Database Management Systems**. Gartner. Stamford. 2015.

GARTNER. **Magic Quadrant for Operational Database Management Systems**. Gartner. Stamford. 2016.

GEORGE, L. **HBase:** The Definitive Guide. Sebastopol: O'Reilly Media, 2011. ISBN 9781449396107.

GILBERT, ; LYNCH, N. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services. **SIGACT News**, New York, v. 33, p. 51-59, Jun 2002.

GOOGLE. Cloud Bigtable. **Google Cloud Platform**, 2005. Disponivel em: <https://cloud.google.com/bigtable/>. Acesso em: 07 Mar 2017.

GOOGLE. Cloud Bigtable. **Google Cloud Plarafform**. Disponivel em: <https://cloud.google.com/bigtable/>. Acesso em: 12 Mar 2017.

GRAY, D. E. **Pesquisa no Mundo Real:** Métodos de Pesquisa. São Paulo: Penso Editora, 2016. ISBN 9788563899293.

GRIEBLER, D. J. **PROPOSTA DE UMA LINGUAGEM ESPECÍFICA DE DOMÍNIO DE PROGRAMAÇÃO PARALELA ORIENTADA A PADRÕES PARALELOS: UM ESTUDO DE CASO BASEADO NO PADRÃO MESTRE/ESCRAVO PARA ARQUITETURAS MULTI-CORE**. PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO GRANDE DO SUL. Porto Alegre, p. 168. 2012.

GUIMARÃES, P. R. B. **Estatística I**. Curitiba: IESDE Brasil S. A., 2010. ISBN 978-85-387-1267-1.

HAN, et al. **Survey on NoSQL database.** In Pervasive computing and applications (ICPCA), 2011 6th international conference. [S.l.]: [s.n.]. 2011. p. 363-366.

HAZELCAST. Hazelcast the Leading In-Memory Data Grid. **Hazelcast**, 2009. Disponivel em: <https://hazelcast.com/>.

HECHT, R.; JABLONSKI,. **NoSQL evaluation:** A use case oriented survey. International Conference on Cloud and Service Computing (CSC). [S.l.]: [s.n.]. 2011. p. 336--341.

HENNESSY, P. J. L.; PATTERSON, D. A. **Organização e Projeto de Computadores:** A Interface Hardware/Software. 4. ed. Rio de Janeiro: Elsevier Brasil, 2014. ISBN 9788535264104.

HERLIHY, M. P.; WING, J. M. Linearizability: A correctness condition for concurrent objects. **ACM Transactions on Programming Languages and Systems (TOPLAS)**, v. 12, n. 3, p. 463-492, 1990.

HINTJENS, P. **ZeroMQ:** Messaging for Many Applications. Sebastopol: O'Reilly Media, 2013. ISBN 978-1-449-33406-2. Disponivel em: <https://books.google.com.br/books?id=KWtT5CJc6FYC&printsec=frontcover&hl=pt-BR>. Acesso em: 17 dez. 2016.

HULLEY, S. B. et al. **Delineando a pesquisa clínica**. 4. ed. Porto Alegre: Artmed Editora, 2015. ISBN 978-85-8271-203-0.

HUPPLER, K. The Art of Building a Good Benchmark. In: NAMBIAR, R.; POESS, M. **Performance Evaluation and Benchmarking**. Berlin: Springer, 2009. p. 18-30. ISBN 978-3-642-10424-4. Disponivel em: <https://books.google.com.br/books?id=zKqgfKoViscC&printsec=frontcover&hl=pt-BR>. Acesso em: 17 dez. 2016.

HURWITZ, J. et al. **Big Data Para Leigos**. Rio de Janeiro: Alta Books Editora, 2016. 328 p. ISBN 9788576089551. Disponivel em: <https://books.google.com.br/books?id=j8hYCwAAQBAJ&dq=big+data&hl=pt-BR&source=gbs_navlinks_s>. Acesso em: 31 out. 2016.

IONESCU, V. M. **The analysis of the performance of RabbitMQ and ActiveMQ**. 2015 14th RoEduNet International Conference - Networking in Education and Research (RoEduNet NER). Craiova: IEEE. 2015. p. 132-137.

JANUSGRAPH. JanusGraph: Distributed Graph Database. **JanusGraph**, 2017. Disponivel em: <http://janusgraph.org/>. Acesso em: 10 abr. 2017.

KABAKUS, A. T.; KARA, R. A performance evaluation of in-memory databases. **Journal of King Saud University - Computer and Information Sciences**, 2016. Disponivel em: <http://dx.doi.org/10.1016/j.jksuci.2016.06.007>. Acesso em: 11 ago. 2016.

KANG, Q. et al. MemTest: A Novel Benchmark for In-memory Database. In: ZHAN, J.; HAN, R.; WENG, C. **Big Data Benchmarks, Performance Optimization, and Emerging Hardware**. Hangzhou: Springer, 2014. p. 34-46. ISBN 978-3-319-13021-7. Disponivel em: <https://books.google.com.br/books?id=LwRNBQAAQBAJ&printsec=frontcover&hl=pt-BR>. Acesso em: 17 dez. 2016.

KIMBALL, ; ROSS,. **The Data Warehouse Toolkit:** The Definitive Guide to Dimensional Modeling. 3. ed. Hoboken: John Wiley & Sons, 2013. ISBN 978-11-187-3228-1.

KLEPPMANN, M. Please stop calling databases CP or AP. **Martin Kleppmann's Blog**, 11 maio 2015. Disponivel em: <https://martin.kleppmann.com/2015/05/11/please-stop-calling-databases-cp-or-ap.html>. Acesso em: 29 mar. 2017.

KREPS, J.; NARKHEDE, N.; RAO, J. Kafka: a distributed messaging system for log processing. **NetDB'11**, Athens, Greece, 12 jun. 2011.

LAKE, P.; CROWTHER, P. **Concise Guide to Databases:** A Practical Introduction. London: Springer, 2013. ISBN 978-1-4471-5601-7.

LARSON, R.; FARBER, B. **Estatística aplicada**. Tradução de Luciane Ferreira. 4. ed. São Paulo: Pearson Prentice Hall, 2010. ISBN 978-85-7605-372-9.

LIANG, F. et al. **Performance benefits of DataMPI:** a case study with BigDataBench. Workshop on Big Data Benchmarks, Performance Optimization, and Emerging Hardware. [S.l.]: Springer. 2014. p. 111--123.

LOESCH, C. **Probabilidade e estatística**. Rio de Janeiro: LTC, 2012. ISBN 978-85-216-2100-3.

LOVATO, A. **Metodologia da Pesquisa**. Três de Maio: SETREM, 2013. 272 p. ISBN 978-85-99020-05-0.

LUBOW, E.; BRADBERRY, R. **Practical Cassandra:** A Developer's Approach. Crawfordsville: Addison-Wesley, 2013. 208 p. ISBN 9780133440218.

MACEDO, N. D. D. **Iniciação a pesquisa bibliográfica:** guia do estudante para a fundamentação do trabalho de pesquisa. 2 ed. ed. São Paulo: Unimarco, 1994. ISBN 85-15-01132-8.

MANNINO, M. V. **Projeto, Desenvolvimento de Aplicações e Administração de Banco de Dados**. 3. ed. Porto Alegre: AMGH Editora, 2008. ISBN 9788580553635.

MARCHAL,. **XML by Example**. Indianapolis: Que Publishing, 2002. ISBN 9780789725042.

MARKLOGIC. Best Database for Integrating Data From Silos | MarkLogic. **MarkLogic**, 2001. Disponivel em: <http://www.marklogic.com/>. Acesso em: 04 mar. 2017.

MARON, C. A. F. **Avaliação e comparação da computação de alto desempenho em ferramentas opensource de administração de nuvem usando estações de trabalho**. Sociedade Educacional Três de Maio - SETREM. Três de Maio, p. 352. 2014.

MARQUES, L. M. O. **Base de Dados em Memória:** Sistemas de indexação. Porto: Instituto Superior de Engenharia do Porto. 2002.

MEMCACHED. memcached - a distributed memory object caching system. **memcached**. Disponivel em: <https://memcached.org/>. Acesso em: 13 nov. 2016.

MICROSOFT. Graph Engine. **Graph Engine**, 2017. Disponivel em: <https://www.graphengine.io/>. Acesso em: 07 mar. 2017.

MONGODB. MongoDB for GIANT ideas. **MongoDB**, 2009. Disponivel em: <https://www.mongodb.com/>. Acesso em: 20 Fev 2017.

MORAES, F. C. C. **Desafios estratégicos em gestão de pessoas**. Curitiba: IESDE, 2012. ISBN 9788538722779.

NAYAK, ; PORIYA, ; POOJARY,. Type of NOSQL Databases and its Comparison with Relational Databases. **International Journal of Applied Information Systems (IJAIS)**, Nova Iorque, v. 5, p. 17-19, Mar 2013. ISSN 2249-0868.

NELSON, J. **Mastering Redis**. Birmingham: Packt Publishing, 2016. ISBN 978-1-78398-818-1. Disponivel em: <https://books.google.com.br/books?id=OANwDQAAQBAJ&printsec=frontcover&hl=pt-BR>. Acesso em: 19 dez. 2016.

NEO TECHNOLOGY. Neo4j, the world's leading graph database - Neo4j Graph Database. **Neo4j**, 2007. Disponivel em: <https://neo4j.com/>. Acesso em: 04 mar. 2017.

NEO4J. **Neo4J**, 2007. Disponivel em: <https://neo4j.com/>. Acesso em: 21 Fev 2017.

O'NEIL, P. et al. **The star schema benchmark and augmented fact table indexing**. Technology Conference on Performance Evaluation and Benchmarking. [S.l.]: Springer. 2009. p. 237-252.

ORACLE. **Oracle Database VLDB and Partitioning Guide, 11g Release 2 (11.2):** E25523-01. Redwood City: Oracle, 2011. Disponivel em: <http://docs.oracle.com/cd/E11882_01/server.112/e25523.pdf>. Acesso em: 17 dez. 2016.

ORACLE MYSQL. MySQL 5.7: 3x Faster. **MySQL**. Disponivel em: <https://www.mysql.com/products/enterprise/database/>. Acesso em: 31 Mar 2017.

ORIENTDB. OrientDB - Distributed Graph/Document Multi-Model Database. **OrientDB**, 2010. Disponivel em: <http://orientdb.com/>. Acesso em: 05 mar. 2017.

OXFORD. **A Dictionary of Computer Science**. 7. ed. New York: Oxford University Press, 2016. ISBN 9780191768125. Disponivel em: <https://books.google.com.br/books?id=MFU0CwAAQBAJ&printsec=frontcover&hl=pt-BR>. Acesso em: 17 dez. 2016.

PARHAMI , B. **Arquitetura de Computadores:** de microprocessadores e supercomputadores. São Paulo: McGraw-Hill, 2007. ISBN 978-85-7726-025-6.

PATTERSON, D. A.; HENNESSY, J. L. **Organização e projeto de computadores:** a interface hardware/software. 4. ed. Rio de Janeiro: Elsevier Brasil, 2014. ISBN 9788535264104.

POKORNY, J. NoSQL databases: a step to database scalability in web environment. **International Journal of Web Information Systems**, v. 9, p. 69--82, 2013.

POLLOCK, J. **Web Semântica Para Leigos**. Rio de Janeiro: Alta Books, 2011. 424 p. ISBN 978-85-7608-465-5.

PRADO, ; SOUZA, C. A. D. **Fundamentos de Sistemas de Informação**. Rio de Janeiro: Elsevier Brasil, 2014. ISBN 9788535274363.

PROJECT VOLDEMORT. Voldemort. **Project Voldemort:** A distributed database. Disponivel em: <http://www.project-voldemort.com/voldemort/>. Acesso em: 17 dez. 2016.

RABBITMQ. RabbitMQ - Messaging that just works. **RabbitMQ**. Disponivel em: <http://www.rabbitmq.com/>. Acesso em: 15 nov. 2016.

RABL, T. et al. Solving Big Data Challenges for Enterprise Application Performance Management. **Proceedings of the VLDB Endowment**, Istanbul, 5, n. 12, ago. 2012. 1724-1735. Disponivel em: <http://vldb.org/pvldb/vol5/p1724_tilmannrabl_vldb2012.pdf>. Acesso em: 17 dez. 2016.

RABL, T. et al. **Towards a complete BigBench implementation**. Workshop on Big Data Benchmarks. [S.l.]: Springer. 2014. p. 3-11.

RAMACHANDRAN, S. lambdazen / bitsy - Bitbucket. **Bitbucket**, 2013. Disponivel em: <https://bitbucket.org/lambdazen/bitsy/>. Acesso em: 07 mar. 2017.

REDIS. Redis. **Redis.io**, 2009. Disponivel em: <http://redis.io/>. Acesso em: 13 nov. 2016.

RÊGO, B. L. **Gestão e Governança de Dados:** Promovendo dados como ativo de valor nas empresas. Rio de Janeiro: Brasport, 2013. ISBN 9788574525891.

ROCKENBACH, D. A. et al. **Estudo Comparativo de Banco de Dados Chave-Valor com Armazenamento em Memória**. ERBD - Escola Regional de Banco de Dados. Passo Fundo: PUC. 2017.

ROWE, ; REIDY, J. G.; DANCEY, C. P. **Estatística Sem Matemática para as Ciências da Saúde:** Métodos de Pesquisa. Porto Alegre: Penso, 2017. 502 p. ISBN 978-85-8429-100-7.

SALMINEN, A.; TOMPA, F. **Communicating with XML**. New York: Springer, 2012. ISBN 978-1-4614-0992-2. Disponivel em: <https://books.google.com.br/books?id=Bg_WcOOe-tIC&printsec=frontcover&hl=pt-BR>. Acesso em: 13 nov. 2016.

SANFILIPPO, S. On Redis, Memcached, Speed, Benchmarks and The Toilet. **antirez weblog**, 21 set. 2010. Disponivel em: <http://antirez.com/post/redis-memcached-benchmark.html>. Acesso em: 16 nov. 2016.

SHAO, B.; WANG, H.; LI, Y. **Trinity:** A distributed graph engine on a memory cloud. Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data. New York: ACM. 2013.

SMITH, B. **JSON básico:** Conheça o formato de dados preferido da web. São Paulo: Novatec Editora, 2015. ISBN 9788575224366.

SNYDER, B.; BOSANAC, D.; DAVIES, R. **ActiveMQ in Action**. Stamford: Manning Publications, 2011. ISBN 978-1-933988-94-8.

SOLIMAN, A. **Getting Started with Memcached**. Birmingham: Packt Publishing, 2013. ISBN 978-1-78216-322-0.

STORCK , L. et al. **Experimentação Vegetal**. Santa Maria: UFSM, 2006. ISBN 9788573910712.

SULLIVAN, D. **NoSQL for Mere Mortals**. New Jersey: Adisson-Wesley, 2015. ISBN 978-0-13-402321-2. Disponivel em: <https://books.google.com.br/books?id=wfnOBwAAQBAJ&printsec=frontcover&hl=pt-BR>. Acesso em: 16 dez. 2016.

SUMBALY, R. et al. Serving large-scale batch computed data with Project Voldemort. **Proceedings of the 10th USENIX conference on File and Storage Technologies**, 2012.

TAVEIRA, L. F. R. **Monitoramento de ambientes computacionais distribuídos em tempo real**. Universidade de Brasília. Brasília, p. 57. 2015.

THORNTON,. Chicago's WindyGrid: Taking Situational Awareness to a New Level. **Data-Smart City Solutions**, 2013. Disponivel em: <http://datasmart.ash.harvard.edu/news/article/chicagos-windygrid-taking-situational-awareness-to-a-new-level-259>. Acesso em: 20 Fev 2017.

TITAN. Titan Distributed Graph Database. **Titan**, 2012. Disponivel em: <http://titan.thinkaurelius.com/>. Acesso em: 21 Fev 2017.

TPC. TPC BENCHMARK™ C - Standard Specification, Revision 5.11. **TPC**, fev. 2010. Disponivel em: <http://www.tpc.org/TPC_Documents_Current_Versions/pdf/tpc-c_v5.11.0.pdf>. Acesso em: 17 dez. 2016.

TPC BENCHMARK™. TPC-H. **TPC™**, 2017. Disponivel em: <http://www.tpc.org/tpch/>. Acesso em: 31 Mar 2017.

TPC BENCHMARK™. TPC-W. **TPC ™**, 2017. Disponivel em: <http://www.tpc.org/tpcw/>. Acesso em: 31 Mar 2017.

TPC BENCHMARK™. TPC-DS. **TPC™**. Disponivel em: <http://www.tpc.org/tpcds/>. Acesso em: 31 Mar 2017.

VIDELA, A.; WILLIAMS, J. J. W. **RabbitMQ in Action:** Distributed Messaging for Everyone. New York: Manning Publications Co., 2012. ISBN 9781935182979.

VIEIRA, S. **Bioestatística Tópicos Avançados**. Rio de Janeiro: Elsevier Brasil, 2011. 288 p. ISBN 978-85-352-5496-9.

VOHRA, D. **Pro Couchbase Development:** A NoSQL Platform for the Enterprise. New York: Apress, 2015. ISBN 978-1-4842-1434-3. Disponivel em: <https://books.google.com.br/books?id=6_BUCgAAQBAJ&printsec=frontcover&hl=pt-BR>. Acesso em: 16 dez. 2016.

WOODSIDE, M.; FRANKS, G.; PETRIU, D. C. **The Future of Software Performance Engineering**. Future of Software Engineering, 2007. FOSE '07. Minneapolis: IEEE. 2007. p. 171-187.

YAHOO. Yahoo Cloud Serving Benchmark | research.yahoo.com. **Yahoo!**, 28 abr. 2010. Disponivel em: <https://research.yahoo.com/news/yahoo-cloud-serving-benchmark/>. Acesso em: 17 dez. 2016.

YAHOO. Yahoo! Cloud System Benchmark (YCSB). **Yahoo! Cloud System Benchmark (YCSB)**, 2011. Disponivel em: <https://github.com/brianfrankcooper/YCSB>. Acesso em: 07 Mai 2017.

YUHANNA, N.; LEGANZA, G.; AUSTIN, C. **The Forrester Wave™: Big Data NoSQL, Q3 2016**. Forrester Research. Cambridge. 2016.

YUHANNA, N.; LEGANZA, G.; WARRIER, S. **The Forrester Wave™: Document Stores, Q3 2016**. Forrester Research. Cambridge. 2016.

ZHANG, M. et al. In-Memory Big Data Management and Processing: A Survey. **IEEE Transactions on Knowledge and Data Engineering**, v. 27, n. 7, p. 1920-1948, Jul 2015.

LISTA DE APÊNDICES

APÊNDICE A – ORÇAMENTO

Para a execução do projeto de foram previstos alguns investimentos financeiros, esta previsão orçamentária está descriminada no Quadro 55. O mesmo está apresentado em 4 colunas, onde a primeira apresenta a descrição do material, a segunda coluna a quantidade prevista, a terceira coluna representa o valor unitário de cada item e a quarta e última coluna o valor total baseado no valor unitário e na quantidade estimada.

Quadro 55 - Previsão orçamentária

Material necessário	**Unidade de Medida**	**Quantidade**	**Valor(R$) unitário**	**Total (R$)**
Capa e encadernação das entregas parciais	Cópia preta	15	0,12	1,80
Encadernação	Capa	1	120,00	120,00
Impressão do projeto	Cópia preta	60	0,12	7,20
Impressão do relatório	Cópia preta	800	0,12	96,00
Deslocamento	Quilômetro	660	0,50	330,00
Horas de pesquisa e desenvolvimento	Horas	440	25,00	11.000,00
Ambiente estimativa	Meses	3	77,00	231,00
Total em R$				**11.786,00**

Para estimativa do valor relacionado ao ambiente, foram avaliados os orçamentos oferecidos pela Amazon AWS e Google Cloud. O ambiente previsto teria 8 núcleos de processamento, 30 GiB de memória RAM, e disco SSD de 375GB.

APÊNDICE B – CRONOGRAMA

O cronograma representado no Quadro 56 e Quadro 57 demonstra as etapas que devem ser percorridos durante o desenvolvimento do projeto. Eles estão estruturados em 2 colunas principais, na primeira estão descritas as atividades, e a segunda coluna principal representa o período em que se pretende executar as atividades. A coluna que representa o período ainda é subdividida em colunas secundárias com os meses nos quais o projeto será desenvolvido.

Quadro 56 - Cronograma de Atividades Propostas 2016

	2016					
Atividades	**Out.**		**Nov.**		**Dez.**	
	1	2	1	2	3	4
Elaboração do projeto		X	X			
Desenvolvimento da justificativa		X	X			
Elaboração da metodologia		X	X			
Conclusão do projeto			X			
Entrega do projeto				X		
Referencial teórico			X	X		
Apresentação do projeto do Trabalho de Conclusão de Curso				X		
Alinhamentos para aplicação do TCC				X	X	
Entrega final do projeto, proposta e declaração de autenticidade					X	
Exame						

Quadro 57 - Cronograma de Atividades Propostas 2017

	2017											
Atividades	**Jan.**		**Fev.**		**Mar.**		**Abr.**		**Mai.**		**Jun.**	
	1	2	1	2	1	2	1	2	1	2	1	2
Continuidade Referencial Teórico.	X	X	X	X	X	X						

Estudo da arte dos *message brokers* e banco de dados em memória.	■	■	X	X	X	X						
Analisar funcionalidades dos *message brokers* e bancos de dados em memória.		■	■		X	X						
Pesquisar os *benchmarks*, definir quais serão aplicados.				■	■							
Preparar o ambiente para aplicação dos *benchmarks.*						■						
Aplicar os *bechmarks* nos *message brokers* e nos bancos de dados em memória avaliados.						■	■	■	■			
Documentar os resultados obtidos						■	■	■	■	■		
Estabelecer a correlação entre as métricas ou indicadores de desempenho.								■	■	■		
Escrever e publicar artigo referente a pesquisa									■	■		
Entrega final do relatório científico.										■		
Apresentação do trabalho de conclusão de curso												■

Legenda:

Previsto: ■

Realizado: X

APÊNDICE C – ARTIGO ESCOLA REGIONAL DE BANCO DE DADOS

Estudo Comparativo de Banco de Dados Chave-Valor com Armazenamento em Memória

Dinei A. Rockenbach[1], Nadine Anderle[1], Dalvan Griebler[1 2], Samuel Souza[1]

[1]Laboratório de Pesquisas Avançadas para Computação em Nuvem (LARCC)

Faculdade Três de Maio (SETREM) – Três de Maio – RS – Brasil

[2]Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS/PPGCC)

Porto Alegre – RS – Brasil

{dineiar,nadianderle}@gmail.com,dalvan.griebler@acad.pucrs.br,

samuel@samuelsouza.com

Abstract. *Key-value databases emerge to address relational databases' limitations and with the increasing capacity of RAM memory it is possible to offer greater performance and versatility in data storage and processing. The objective is to perform a comparative study of key-value databases with memory storage Redis, Memcached, Voldemort, Aerospike, Hazelcast and Riak KV. Thus, the work contributed to an analysis of different databases and with results that qualitatively demonstrated the characteristics and pointed out the main advantages.*

Resumo. *Bancos de Dados (BD) chave-valor surgem para suprir limitações de BDs relacionais e com o aumento da capacidade das memórias RAM é possível oferecer maior desempenho e versatilidade no armazenamento e processamento dos dados. O objetivo é realizar um estudo comparativo dos BDs chave-valor com armazenamento em memória Redis, Memcached, Voldemort, Aerospike, Hazelcast e Riak KV. Assim, o trabalho contribuiu para uma análise de diferentes BDs e com resultados que demonstraram qualitativamente as características e apontaram as principais vantagens.*

7. Introdução

As necessidades de alta disponibilidade e velocidade, bem como o aumento e consumo de dados nos sistemas e aplicações atuais, influenciaram no desenvolvimento de novas formas para o armazenamento de dados. Estas vão além dos bancos de dados relacionais, impulsionando o movimento NoSQL (*Not only SQL*) [Fowler 2015]. Não obstante, com a popularidade em alta em um mercado sem um líder estabelecido, há uma consequente explosão no número de sistemas de armazenamento disponíveis, o que dificulta a tomada de decisão quanto à opção que melhor supre as necessidades organizacionais.

Com o objetivo de solucionar problemas específicos, os bancos NoSQL foram categorizados de acordo com suas características e otimizações. Por exemplo, a Amazon utiliza seu sistema chave-valor (key-value) Dynamo [DeCandia et al. 2007] para gerenciar as listas de mais vendidos, carrinhos de compras, preferências do consumidor, gerenciamento de produtos, entre outras aplicações. Exitem também sistemas de família de colunas (column family ou columnar) que foram influenciados pelo Bigtable da Google [Chang et al. 2008]. Enquanto isso, os assim chamados de sistemas de documentos (document) resultaram, por exemplo, no MongoDB. Por fim, do chamado banco triplo (*graph database* ou *triple*), tem-se como exemplo o Neo4j. Cada uma destas categorias traz sistemas que cobrem diferentes limitações dos bancos relacionais tradicionais.

Este artigo está organizado em 5 seções, incluindo esta seção introdutória. Na Seção 2 estão os trabalhos relacionados. A Seção 3 traz o embasamento sobre os bancos de dados pesquisados. Na Seção 4 está detalhado o estudo comparativo destes sistemas. Por fim, na Seção 5 estão as conclusões e propostas para trabalhos futuros.

8. Trabalhos Relacionados

Nesta seção é apresentada uma discussão sobre os trabalhos relacionados publicados recentemente na literatura. De forma semelhante, todos os trabalhos escolhidos buscam caracterizar e comparar bancos de dados NoSQL. No entanto, não voltam os estudos para um determinado tipo de banco de dados, o que é importante para avaliar características específicas. Tanto [Hecht and Jablonski 2011] quanto [Han et al. 2011] se propõem a fazer um estudo e avaliação dos bancos de dados NoSQL, com o mesmo objetivo principal: prover informações para auxiliar na escolha do banco NoSQL que melhor atende às necessidades. Por não delimitarem um tipo específico de banco NoSQL, ambos possuem um escopo mais abrangente do que o presente trabalho. No ponto de intersecção entre este trabalho e os citados, [Hecht and Jablonski 2011] inclui em seu trabalho os bancos Project Voldemort, Redis e Membase, enquanto que [Han et al. 2011] avalia Redis, Tokyo Cabinet-Tokyo Tyrant e Flare.

Em [Deka 2014] é apresentada uma visão geral de vários sistemas NoSQL e os representantes chave-valor incluídos na avaliação são Hypertable, Voldemort, Dynomite, Redis e Dynamo. Nota-se a falta, porém, de uma visão comparativa mais clara sobre aspectos de garantias de durabilidade, disponibilidade, protocolos suportados, e outras informações que podem vir a ter uma influência significativa na escolha de um banco de dados chave-valor. Já [Zhang et al. 2015] traz uma visão bem estruturada dos objetivos que nortearam o projeto de cada um dos sistemas descritos no trabalho, o qual foca em sistemas com gerenciamento e processamento de dados em memória. Dentre os sistemas estudados, os representantes dos bancos chave-valor são MemepiC, RAMCloud, Redis, Memcached, MemC3 e TxCache. Dos sistemas, o trabalho descreve as cargas de dados mais adequadas ao sistema, a estratégia para construção de índices, o controle de concorrência, tolerância a falhas, tratamentos para conjuntos de dados maiores do que a memória disponível e o suporte a consultas personalizadas

em baixo nível (como stored procedures e scripts em linguagem nativa, por exemplo), porém com pouca abordagem de alto nível que auxilie na escolha de um sistema em favor de outro.

Ainda que o tema NoSQL tenha sido bastante explorado na academia, e a bibliografia focada no armazenamento de dados em memória tenha crescido muito nos últimos anos, nota-se a falta de um estudo comparativo entre os sistemas chave-valor em memória Redis, Memcached, Voldemort, Aerospike, Hazelcast e Riak KV.

9. Banco de Dados com Armazenamento em Memória

O crescimento dos bancos de dados com armazenamento de dados em memória (IMDB ou in-memory databases) segue uma tendência que teve seu início no hardware, com a capacidade da memória dobrando em média a cada três anos e seu preço caindo uma casa decimal a cada cinco anos [Zhang et al. 2015]. As memórias não-voláteis (NVM ou Non-Volatile Memory) como o SSD (Solid State Disk), também têm evoluído, mas seu custo [Kasavajhala 2011], durabilidade e confiabilidade [Schroeder et al. 2016] continuam sendo impeditivos para a maioria das aplicações.

A vantagem em manter os dados na memória ao invés do disco está relacionada à latência de acesso a estes dados, pois remove-se a necessidade de acessar a camada mais lenta da hierarquia de memória, conforme demonstrado pela Figura 1 (adaptada de [Zhang et al. 2015]), que detalha as camadas de armazenamento, bem como uma estimativa de sua capacidade atual e da latência de acesso aos dados nela armazenados.

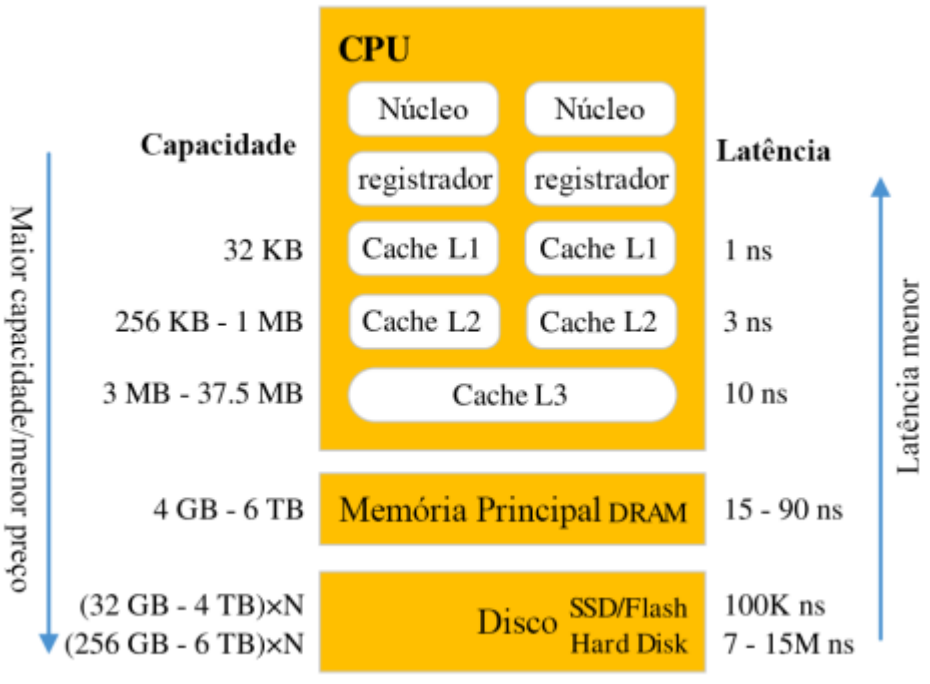


Figura 1. Hierarquia de memória.

Para que os dados possam ser processados pela CPU é necessário que estes estejam nos registradores e para tal é preciso que estes dados passem por todas as camadas da hierarquia de memória até chegarem aos registradores [Zhang et al. 2015]. Como pode ser visto na Figura 1, o disco é a camada mais distante e mais lenta, porém com a maior capacidade de armazenamento. Com o aumento da capacidade da camada de memória principal (composta

pela memória RAM) os IMDB buscam trazer os dados para esta camada e evitar o nível mais lento da hierarquia de memória.

Dentre a miríade de bancos de dados com armazenamento em memória, os bancos chave-valor podem ser considerados os representantes mais versáteis, simples e com melhor desempenho, advindo principalmente da sua simplicidade [Pokorny 2013]. Por tanto, muitos sistemas desta categoria sacrificam garantias de consistência em favor do desempenho [DeCandia et al. 2007] [Fowler 2015]. Nestes bancos, cada valor armazenado está vinculado a uma chave que identifica unicamente um valor [Han et al. 2011], sendo que este valor pode ser tanto um conteúdo binário quando uma estrutura de dados lexa, conforme as funcionalidades oferecidas pelo banco [Pokorny 2013]. Nas próximas seções são apresentados e discutidos os sistemas de armazenamento chave-valor em memória Redis, Memcached, Voldemort, Aerospike, Hazelcast e Riak KV.

9.1 Redis

Redis (REmote DIctionary Server) [Redis 2009] é um sistema de armazenamento de da dos estruturados em memória que pode ser utilizado como banco de dados, cache e mes sage broker [Cao et al. 2016]. Ele opera em um modelo cliente-servidor através de cone xões TCP utilizando um protocolo próprio chamado RESP (REdis Serialization Protocol).

O modelo de dados do Redis é composto por cinco estruturas de dados diferentes para os valores (*string, list, set, sorted set* e *hash*), a persistência dos dados da memória em disco através de dois métodos (*snapshopts* chamados RDB e append-only file ou AOF) [Zhang et al. 2015]. A possibilidade de escalabilidade horizontal através do Redis Cluster foi adicionada apenas em 2015, na versão 3.0 do sistema. Segundo os autores de [Sanfilippo 2010], um sistema deve ser eficiente em um único nó quando for escalado.

9.2 Memcached

O Memcached [Memcached 2003] caracteriza-se como um sistema genérico de cache em memória. Ele foi construído pensando na melhoria de desempenho de aplicações web através da redução na demanda de requisições ao banco de dados em disco. Brad Fitzpatrick desenvolveu ele para melhorar o desempenho do site Livejournal.com através de uma solução melhor de cache [Galbraith 2009]. A sua implementação é na lingua gem Perl e posteriormente reescrito em C. O Memcached utiliza uma arquitetura multi thread e o controle de concorrência interno é feito através de uma hash-table estática de locks [Zhang et al. 2015].

A classificação do Memcached como banco de dados é discutível, uma vez que o mesmo não implementa persistência, failover [Galbraith 2009] nem escalabilidade ho rizontal, pois a distribuição dos dados entre múltiplas instâncias do sistema deve ser feita pelo cliente [Zhang et al. 2015]. O funcionamento do Memcached segue o modelo cliente-servidor e a comunicação ocorre através de conexões TCP ou UDP utilizando um protocolo próprio que suporta textos puros em ASCII ou dados binários [Soliman 2013].

9.3 Voldemort

O Voldemort [Voldemort 2009] foi desenvolvido pelo LinkedIn em linguagem Java com o objetivo de gerenciar funcionalidades dependentes de associações entre dados da rede social, tais como a recomendação de relacionamentos através da análise dos relaciona mentos atuais [Sumbaly et al. 2012].

O Voldemort é inspirado no Dynamo, da Amazon [DeCandia et al. 2007], oferece comandos simples (*put, get* e *delete*) [Sumbaly et al. 2013] e uma arquitetura completa mente distribuída, onde cada nó é independente e não existe um servidor principal de coordenação [Deka 2014]. O sistema é completamente modularizado e tanto a seriali zação dos dados quanto a persistência são oferecidas através de módulos plugáveis. Se gundo [Sumbaly et al. 2012], a grande vantagem do Voldemort em relação ao Dynamo, é um mecanismo próprio de armazenamento desenhado para o pré-carregamento de grandes volumes de dados, em que o Voldemort passa a funcionar em modo somente leitura.

9.4 Aerospike

O Aerospike [Aerospike 2012] tem uma arquitetura modelada com foco em velocidade na análise de dados, escalabilidade e confiabilidade para aplicações web. Esse banco dedados se apresenta como uma solução para a combinação de diferentes tipos de dados e também acessos por milhares de usuários. Pensando nisso, as suas operações são focadas em chave-valor e otimizadas para o uso da memória RAM em conjunto com memórias flash (NVM) [Aerospike 2012].

Diferente de vários de seus concorrentes, o próprio Aerospike disponibiliza bibliotecas de integração aos clientes, a fim de melhorar o desempenho na sua utilização. Quanto ao cluster, todos os nós são iguais, em uma arquitetura conhecida como shared nothing. A respeito do server é possível utilizar índices secundários e definir funções para otimizar a utilização dos dados. E por fim, a camada de armazenamento incorpora a utilização da memória RAM e de sistemas de armazenamento permanente.

9.5 Hazelcast

O Hazelcast [Hazelcast 2009] é uma ferramenta distribuída sob licença open source e comercial desenvolvida em Java. Possui seu foco em computação distribuída e escalabili dade horizontal, se destacando dos concorrentes por oferecer as garantias ACID (Atomi cidade, Consistência, Isolamento e Durabilidade) dos bancos de dados relacionais tradicionais.

O cluster funciona em uma arquitetura *shared nothing*, onde não existe um ponto único de falha. Além de oferecer clientes para as linguagens comuns como Java, C, C++ e C#, o Hazelcast oferece uma API REST, está preparado para trabalhar com o protocolo de comunicação do Memcache e pode ser utilizado através do Hibernate [Hazelcast 2009].

9.6 Riak KV

O Riak KV [Basho 2009] possui como principal objetivo oferecer disponibilidade máxima, com escalabilidade horizontal em forma de cluster, sendo considerado um banco de dados de simples operação e fácil escalabilidade. Em sua versão comercial há suporte a *multi-cluster replication*, ou seja, é possível realizar a replicação de dados através de diferentes clusters, geograficamente distantes, a fim de reduzir a latência de acesso uniformemente para clientes através do globo.

Nota-se claramente a influência do Dynamo [DeCandia et al. 2007] no Riak KV, desde suas funcionalidades para execução distribuída até nas configurações do fator de replicação e arquitetura *shared nothing*.

10. Estudo comparativo

Esta seção apresenta uma comparação entre os sistemas apresentados anteriormente. Para isso, as características foram classificadas em três grupos: (I) características mercadológicas, onde são explorados aspectos sem relação direta com funcionalidades ou o funcionamento do banco, tais como ano de lançamento, licenciamento e linguagem de desenvolvimento; (II) características do projeto, onde são descritas definições decididas no projeto do sistema, tais como escalabilidade, disponibilidade e consistência; e (III) características de manutenção, onde são explanadas os aspectos que tem relação direta com a manutenção e suporte ao sistema, tais como ferramentas internas para monitoramento e interface de gerenciamento.

Na Tabela 1 é possível avaliar: o ano em que a primeira versão do sistema foi lançada, os licenciamentos sob os quais o software é distribuído, a linguagem na qual o sistema foi desenvolvido, os sistemas operacionais suportados, as linguagens nas quais são oferecidos clientes para comunicação e os protocolos de comunicação suportados. A partir dos dados disponibilizados, os interessados podem avaliar a maturidade do sistema, se a licença está alinhada com as necessidades empresariais e se a infraestrutura disponível e o esforço de implementação estão dentro do esperado.

Vale ressaltar que o Redis não suporta oficialmente Windows, mas uma versão para Windows x64 é mantida pela equipe da MS Open Tech (Microsoft Open Technolo gies). Quanto ao Voldemort e ao Hazelcast, como os mesmos rodam na JVM (Java Virtual Machine), podem-se considerar os sistemas operacionais suportados por esta tecnologia. Tanto Aerospike quanto Riak KV oferecem pacotes para sistemas baseados nas distribui- ções Linux Red Hat, Debian e Ubuntu. O Aerospike ainda oferece sua execução no OS X e Windows através de máquinas virtuais.

Tabela 1. Características mercadológicas

	Redis	Memcached	Voldemort	Aerospike	Hazelcast	Riak KV
Lançamento	2009	2003	2009	2012	2009	2009
Licenciamento	BSD-3 e comercial	BSD-3	Apache 2	AGPL e comercial	Apache 2 e comercial	Apache 2 e comercial
Desenvolvido	C	C	Java	C	Java	Erlang
SO Suporte	Linux, BSD, OSX e Windows	Debian/Ubuntu e Windows	JVM	Linux, OS X e Windows	JVM	Linux
Clientes	48 linguagens	Não existe listagem oficial	4 linguagens	12 linguagens	6 linguagens	21 linguagens
Protocolos	Próprio (RESP)	Próprio	HTTP, Socket, NIO	Próprio e JDBC	Próprio e Memcached	API HTTP e próprio

Quanto ao item linguagens com cliente, é importante notar que foram considera dos apenas as linguagens e clientes listados no site oficial de cada sistema e que o site do Memcached não oferece uma listagem oficial das linguagens suportadas. Nota-se também que as linguagens C++, Java e Python são as únicas para as quais todos os sistemas possuem clientes. Os protocolos de comunicação são o meio de comunicação entre o cliente e o servidor, porém, na maioria dos casos a comunicação é feita através de um dos clientes já construídos e a empresa não precisa se preocupar com o protocolo utilizado pelo cliente

Na Tabela 2 é possível avaliar: as opções para escalabilidade horizontal (ou clusterização), a classificação do sistema segundo o teorema CAP [Brewer 2000], como é feito o controle de concorrência, o suporte à transações ACID, as opções de persistência dos dados em disco, o suporte a dados complexos e as opções para autenticação do cliente. Analisando a

Tabela 2 é possível avaliar se as funcionalidades e características do banco de dados atendem às demandas e características referentes aos dados que se pretende armazenar nos mesmos.

Quanto à escalabilidade, todos os bancos exceto o Memcached incluem suporte a *sharding* e replicação, sendo que a maioria (a exemplo do Dynamo [DeCandia et al. 2007]) oferecem fator de replicação configurável para leituras e escritas. O Redis utiliza o modelo *master/slave*, comumente utilizado nas bases de dados relacionais tradicionais.

O teorema CAP (*Consistency, Availability e Partition tolerance*) foi proposto por Eric Brewer em [Brewer 2000] e verificado em [Gilbert and Lynch 2002], desde então passou a ser largamente aceito pela academia. O teorema afirma que na existência de uma falha de comunicação (*partition*) cada nó de um sistema distribuído deve escolher entre responder requisições, mantendo a disponibilidade (*availability*) e assumindo o risco de não retornar os dados mais atuais, ou rejeitar requisições para garantir a consistência dos dados (*consistency*). Sistemas classificados como AP priorizam a disponibilidade, enquanto que sistemas classificados como CP priorizam a consistência. O teorema tem sido alvo de muitas críticas e Brewer explora algumas de suas limitações em [Brewer 2012], enquanto Abadi propõe o teorema PACELC como alternativa em [Abadi 2012].

Quanto à classificação do Redis, é importante mencionar que o ele não atende todos os requisitos de um sistema CP de [Brewer 2000], por usar replicação assíncrona entre os nós do cluster. O teorema CAP também não se aplica ao Memcached pelo fato de ele não suportar a criação de clusters e, portanto, não haver comunicação entre nós. O Riak KV possui uma configuração onde é possível definir qual dos atributos devem ser preservados (disponibilidade ou consistência).

Tabela 2. Características do projeto

	Redis	Memcached	Voldemort	Aerospike	Hazelcast	Riak KV
Escalabilidade horiontal	Mestre - Escravo	Não	Fator de Replicação	Fator de Replicação	Fator de Replicação	Fator de Replicação
Teorema CAP	CP	N/A	AP	AP	AP	Config.
Controle Concorrência	Single thread	Mutex lock	MVCC	Test-and-set	Multi-single-thread	MVCC
Transações	Parcial	Não	Não	Parcial	Sim	Não
Persistência em disco	RDB e AOF	Não	Config.	Assínc.	Banco auxi-liar	Banco auxiliar
Suporte a dados complexos	Sim	Não	Sim	Sim	Sim	Sim
Autenticação	Simples	SASL	Kerberos	Somente comercial	Simples, SSL, Kerberos,IP	Sim, e autorização

O controle de concorrência é implementado de diferentes maneiras. No Redis a execução é single-thread e as requisições são processadas de forma assíncrona in ternamente [Zhang et al. 2015], sendo que apenas um *master* responde por uma deter minada chave, portanto não há concorrência. O Memcached utiliza *mutex* (*mutual exclusive*) *lock*. Tanto Voldemort quanto Riak KV seguem a implementação do Dynamo [DeCandia et al. 2007] e utilizam *vector clocks*, uma implementação do versionamento baseado em *locking* otimista conhecida por MVCC (Multi Version Concurrency Control). O Aerospike utiliza o método conhecido como *test-and-set* ou *check-and-set* (CAS), uma operação atômica implementada a baixo nível que escreve em um local de memória e retorna o valor antigo. No Hazelcast, é criada uma *thread* para atender cada uma das par tições internas de dados, portanto ainda que ele seja *multi-thread*, uma chave específica está num contexto single-thread e, portanto, não há concorrência

Quanto as transações, há um nível bastante variado de suporte oferecido pelos sistemas. Ainda que o Redis tenha suporte básico a transações, estas não possuem opção de *rollback* e a durabilidade da mesma depende da persistência em disco. Memcached, Voldemort e Riak KV declaram não suportarem transações, enquanto que o Aerospike suporta transações que envolvam uma única chave ou que sejam somente leitura, no caso de envolverem múltiplas chaves. O Hazelcast se destaca sendo o único a oferecer transações ACID completas.

A persistência em disco é oferecida por todos os bancos, exceto o memcached. No Redis são oferecidas duas formas complementares: snapshot (RDB), onde todos os dados na memória são gravados em disco, e append-only file (AOF), onde cada operação é gravada em um arquivo de log e o arquivo é reescrito quando chega em um tamanho pré-determinado. O Voldemort oferece opções para configurar a persistência como síncrona (*write through*, onde a operação é persistida antes do retorno ao cliente) ou assíncrona (*write behind*, onde o cliente recebe a confirmação e posteriormente a operação é persistida), enquanto que o Aerospike faz a persistência de forma assíncrona. Vale mencionar que o Aerospike tem melhorias focadas no uso de SSD como dispositivo de armazenamento permanente. Tanto Hazelcast como Riak KV oferecem persistência através do acoplamento de um banco de dados auxiliar e a sincronicidade é configurável.

Referente ao suporte a dados complexos, vale notar que todos os sistemas, exceto o Memcached, suportam chaves do tipo lista e hashtable (ainda que com nomes diferentes). Hazelcast e Voldemort se baseiam fortemente nas classes do Java, Redis e Riak KV ainda oferecem suporte ao tipo HyperLogLogs, enquanto Redis e Aerospike oferecem suporte a tipagem ou comandos relativos a georreferenciamento.

Finalizando, enquanto que Redis oferece autenticação simples através de credenciais pré-configuradas, o Memcached oferece autenticação através do protocolo SASL(*Simple Authentication and Security Layer*). O Voldemort é integrado ao protocolo Kerberos. O Aerospike oferece autenticação apenas em sua versão com licenciamento comercial. O Hazelcast oferece todos os protocolos supracitados e o SSL. Por último, o Riak KV oferece um sistema próprio de usuários e grupos, com autenticação e autorização baseada em diversos mecanismos, incluindo senhas e certificados digitais.

Na Tabela 3 está descrita a existência de interfaces de gerenciamento, ferramentas de monitoramento e benchmarks para os sistemas estudados. É importante notar que fo ram avaliadas apenas as ferramentas oficiais dos desenvolvedores dos sistemas. Portanto, muitas destas ferramentas, ainda que não estejam declaradas aqui, já foram desenvolvidas pela comunidade e estão disponíveis. Estas informações são particularmente interessantes para avaliar o esforço de manutenção que será despendido após a implantação do sistema.

Tabela 3. Características de manutenção

	Redis	Memcached	Voldemort	Aerospike	Hazelcast	Riak KV
Interface de Gerenciamento	Não	Não	Básica	À parte	Comercial	Sim
Ferramentas de Monitoramento	‘INFO’	‘stats’	JMX	‘asadm’	JMX	‘stats’
Benchmark embutido	Sim	Não	Sim	Sim	Não	Sim

Quanto às interfaces de gerenciamento oferecidas pelos sistemas avaliados, o Redis e o Memcached são os únicos que não as oferecem nativamente (ainda que existam opções na comunidade) e o Voldemort oferece uma interface básica à parte escrita em Ruby, que está sem manutenção. O Aerospike oferece uma interface que deve ser instalada à parte. No Hazelcast, esta funcionalidade está disponível apenas na versão comercial. O Riak KV é o único onde a interface de gerenciamento já está integrada ao código principal do programa, não requerendo nenhuma instalação extra.

A respeito de ferramentas de monitoramento, o Redis oferece comandos como INFO, MEMORY e LATENCY, enquanto que o Memcached oferece o comando stats e o Aerospike oferece o comando asadm. O Voldemort possui uma interface completa de monitoramento exposta através de Java Management Extensions (JMX). Esta é a mesma estratégia utilizada pelo Hazelcast. O Riak KV oferece os comandos stat e stats em sua interface de linha de comando (CLI) riak-admin e a URL /stats em sua API HTTP.

No item benchmark embutido foi avaliado se são disponibilizados benchmarks junto com o sistema, cujo principal objetivo é avaliar o desempenho do sistema em deter minada infraestrutura. Enquanto que Memcached e Hazelcast não oferecem ferramentas próprias para realização de benchmark, no Redis existe a ferramenta redis-benchmark e o Voldemort oferece a *voldemort-performance-tool*. No Aerospike os *benchmarks* estão nos clientes disponibilizados e no Riak KV o nome dado ao benchmark é Basho Bench.

Após comparar as características dos bancos de dados chave-valor com armazenamento em memória, Redis, Memcached, Voldemort, Aerospike, Hazelcast e Riak KV, é fácil entender o motivo pelo qual o Memcached não é considerado um banco de dados, pois seu foco destoa bastante de seus semelhantes. É possível perceber também que, mesmo que o Redis tenha oferecido suporte à clusterização em suas versões mais recentes, os outros sistemas ainda estão à frente quando o assunto é funcionalidades para execu- ção distribuída. É possível perceber também como Voldemort e Hazelcast se utilizam do ecossistema Java para prover funcionalidades interessantes e como o *paper* do Dynamo [DeCandia et al. 2007] influencia principalmente Voldemort e Riak KV.

11. Conclusões

Após estudar os bancos chave-valor com armazenamento em memória, é possível notar que mesmo um subconjunto específico de bancos NoSQL traz muitas variáveis. Neste sentido, destaca-se a grande quantidade de características que devem ser cuidadosamente avaliadas pelo analista para a correta tomada de decisão quanto ao banco mais adequado às necessidades. Dentre estas características, destacam-se o padrão de busca e gravação de dados da aplicação cliente, a importância da durabilidade dos dados, o comportamento desejado frente a partições no cluster, o ambiente de infraestrutura onde a solução será implantada, o ambiente de desenvolvimento da aplicação cliente e as perspectivas de crescimento na demanda da aplicação.

Com as características dos sistemas esclarecidas, percebe-se que outro fator importante para a escolha do banco de dados chave-valor com armazenamento em memória a ser adotado é o desempenho, que é justamente o ponto que traz mais interesse a esta categoria de sistemas. Como trabalho futuro, propõe-se uma avaliação do desempenho dos sistemas aqui estudados, comparando o desempenho das variadas características compartilhadas pelos mesmos.

12. Referências

[Abadi 2012] Abadi, D. (2012). Consistency Tradeoffs in Modern Distributed Database System Design: CAP is Only Part of the Story. Computer, 45(2):37–42.

[Aerospike 2012] Aerospike (2012). Aerospike | High Performance NoSQL Database. Access on <http://www.aerospike.com/>.

[Basho 2009] Basho (2009). Riak KV. Access on <http://basho.com/products/riak-kv/>

[Brewer 2000] Brewer, E. (2000). Towards Robust Distributed Systems. In Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing, PODC ’00, pages 7–, New York, NY, USA. ACM.

[Brewer 2012] Brewer, E. (2012). CAP twelve years later: How the "rules"have changed. Computer, 45(2):23–29

[Cao et al. 2016] Cao, W., Sahin, S., Liu, L., and Bao, X. (2016). Evaluation and Analysis of In-Memory Key-Value Systems. In 2016 IEEE International Congress on Big Data (BigData Congress), pages 26–33.

[Carlson 2013] Carlson, J. L. (2013). Redis in Action. Manning, Shelter Island, NY, USA.

[Chang et al. 2008] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E. (2008). Bigtable: A distributed storage system for structured data. ACM Trans. on Computer Systems (TOCS), 26(2):4.

[DeCandia et al. 2007] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. (2007). Dynamo: amazon's highly available key-value store. ACM SIGOPS operating systems review, 41(6):205–220.

[Deka 2014] Deka, G. C. (2014). A survey of cloud database systems. IT Professional, 16(2):50–57.

[Fowler 2015] Fowler, A. (2015). NoSQL For Dummies. John Wiley & Sons, 111 River Street, Hoboken, New Jersey, USA.

[Galbraith 2009] Galbraith, P. (2009). Developing Web Applications with Apache, MySQL, memcached, and Perl. John Wiley & Sons.

[Gilbert and Lynch 2002] Gilbert, S. and Lynch, N. (2002). Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services. SIGACT News, 33(2):51–59.

[Han et al. 2011] Han, J., E, H., Le, G., and Du, J. (2011). Survey on NoSQL database. In 2011 6th International Conference on Pervasive Computing and Applications, pages 363–366.

[Hazelcast 2009] Hazelcast (2009). Hazelcast the Leading In-Memory Data Grid - Hazelcast.com. Access on <https://hazelcast.com/>.

[Hecht and Jablonski 2011] Hecht, R. and Jablonski, S. (2011). NoSQL evaluation: A use case oriented survey. In 2011 International Conference on Cloud and Service Computing (CSC), pages 336–341.

[Kasavajhala 2011] Kasavajhala, V. (2011). Solid State Drive vs. Hard Disk Drive Price and Performance Study. Proc. Dell Technical White Paper, pages 8–9.

[Memcached 2003] Memcached (2003). memcached - a distributed memory object caching system. Access on <https://memcached.org/>.

[Pokorny 2013] Pokorny, J. (2013). NoSQL databases: a step to database scalability in web environment. International Journal of Web Information Systems, 9(1):69–82.

[Redis 2009] Redis (2009). Redis.io. Access on <http://redis.io/>.

[Robbins 2008] Robbins, S. (2008). RAM is the new disk... Access on <https://www.infoq.com/news/2008/06/ram-is-disk>.

[Sanfilippo 2010] Sanfilippo, S. (2010). On Redis, Memcached, Speed, Benchmarks and The Toilet . Access on <http://antirez.com/post/redis-memcached-benchmark.html>.

[Schroeder et al. 2016] Schroeder, B., Lagisetty, R., and Merchant, A. (2016). Flash Reliability in Production: The Expected and the Unexpected. In Proceedings of the 14th USENIX

Conference on File and Storage Technologies (FAST ’16), FAST ’16, pages 67–80, Santa Clara, CA, USA.

[Soliman 2013] Soliman, A. (2013). Getting Started with Memcached. Packt.

[Sumbaly et al. 2012] Sumbaly, R., Kreps, J., Gao, L., Feinberg, A., Soman, C., and Shah, S. (2012). Serving large-scale batch computed data with Project Voldemort. In Proceedings of the 10th USENIX conference on File and Storage Technologies, page 18.

[Sumbaly et al. 2013] Sumbaly, R., Kreps, J., and Shah, S. (2013). The Big Data Ecosystemat LinkedIn. In Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, SIGMOD ’13, pages 1125–1134, New York, NY, USA. ACM.

[Voldemort 2009] Voldemort (2009). Project Voldemort. Access on <http://www.projectvoldemort.com/>.

[Zhang et al. 2015] Zhang, H., Chen, G., Ooi, B. C., Tan, K.-L., and Zhang, M. (2015). In-Memory Big Data Management and Processing: A Survey. IEEE Transactions on Knowledge and Data Engineering, 27(7):1920–1948.

APÊNDICE D – CÓDIGO FONTE DO PROGRAMA DE MONITORAMENTO

Arquivo utils.h

```
#ifndef UTILS_H_INCLUDED
#define UTILS_H_INCLUDED

#include <cstdio>
#include <iostream>
#include <memory>
#include <string>
#include <stdexcept>
#include <array>
//for file_exists
#include <sys/stat.h>
//for trim
#include <algorithm>
#include <functional>
#include <cctype>
#include <locale>
//for split
// #include <iostream>
// #include <string>
#include <sstream>
// #include <algorithm>
#include <iterator>

using namespace std;

std::string get_prefix() {
    return to_string(time(nullptr)) + "\t";
}

//from http://stackoverflow.com/a/478960/3136474
std::string exec(const char* cmd) {
    std::array<char, 128> buffer;
    std::string result;
    std::shared_ptr<FILE> pipe(popen(cmd, "r"), pclose);
    if (!pipe) throw std::RunTime_error("popen() failed!");
    while (!feof(pipe.get())) {
        if (fgets(buffer.data(), 128, pipe.get()) != NULL)
            result += buffer.data();
    }
    return result;
}

//from http://stackoverflow.com/a/12774387/3136474
inline bool file_exists(const std::string& name) {
```

```
  struct stat buffer;
  return (stat (name.c_str(), &buffer) == 0);
}

//from http://stackoverflow.com/a/217605/3136474
static inline void ltrim(std::string &s) {
    s.erase(s.begin(), std::find_if(s.begin(), s.end(),
            std::not1(std::ptr_fun<int, int>(std::isspace))));
}
static inline void rtrim(std::string &s) {
    s.erase(std::find_if(s.rbegin(), s.rend(),
            std::not1(std::ptr_fun<int, int>(std::isspace))).base(),
s.end());
}
static inline void trim(std::string &s) {
    ltrim(s);
    rtrim(s);
}

//adapted from http://stackoverflow.com/a/236803/3136474
template<typename Out>
void splitNl(const std::string &s, Out result) {
    std::stringstream ss;
    ss.str(s);
    std::string item;
    while (std::getline(ss, item)) {
        *(result++) = item;
    }
}
std::vector<std::string> splitNl(const std::string &s) {
    std::vector<std::string> elems;
    splitNl(s, std::back_inserter(elems));
    return elems;
}

#endif
```

Arquivo monitor.h

```
#ifndef MONITOR_H_INCLUDED
#define MONITOR_H_INCLUDED

#include <cstdio>
#include <iostream>
#include <string>
#include <utils.h>

using namespace std;

class Monitor {
    public:
        std::ofstream fstream;
        std::string file;
        std::string cmd;

        Monitor();
        // Monitor(std::string filename); //I had problems with this guy

        void init(std::string filename, std::string command);
```

```
        void read_status();
        void flush();
        void close();
};

Monitor::Monitor() { };
void Monitor::init(std::string filename, std::string command) {
    file = filename;
    cmd = command;
    // fstream.open(filename, std::ofstream::trunc | std::ofstream::app);
    fstream.open(filename);
};

void Monitor::read_status() {
    fstream << get_prefix() << exec(cmd.c_str());
};

void Monitor::flush() {
    fstream.flush();
};

void Monitor::close() {
    fstream.close();
};

#endif
```

Arquivo monitoring.h

```
#ifndef MONITORING_H_INCLUDED
#define MONITORING_H_INCLUDED

#include <string>
#include <monitor.h> //also includes utils.h

using namespace std;

class Monitoring {
    public:
        bool redis, riak;
        Monitor mon_iostat, mon_free, mon_df, mon_redis, mon_riak;

        //Init system monitoring
        Monitoring();
        Monitoring(std::string out_folder);

        void init(std::string out_folder);
        void detect();
        void flush();
        void close();
};

Monitoring::Monitoring() { };
Monitoring::Monitoring(std::string out_folder) {
    redis = false;
    riak = false;
    mon_iostat.init(out_folder + "iostat.txt", "iostat -c -d -m -x -y");
    mon_free.init(out_folder + "free.txt", "free -m -t");
    mon_df.init(out_folder + "df.txt", "df -T -l");
```

```
    detect();
    if (redis) {
        mon_redis.init(out_folder + "redis.txt", "redis-cli info all");
    }
    if (riak) {
        mon_riak.init(out_folder + "riak.txt", "sudo riak-admin status");
    }
};

//Detect running databases
void Monitoring::detect() {
    //Redis
    redis = false;
    std::string cmd = "redis-cli ping";
    std::string expected = "PONG\n";
    if (exec(cmd.c_str()) == expected) {
        redis = true;
    }

    //Riak
    riak = false;
    cmd = "sudo riak ping";
    expected = "pong\n";
    if (exec(cmd.c_str()) == expected) {
        riak = true;
    }
};

//Free resources
void Monitoring::flush() {
    mon_iostat.flush();
    mon_free.flush();
    mon_df.flush();
    mon_redis.flush();
    mon_riak.flush();
}

//Free resources
void Monitoring::close() {
    mon_iostat.close();
    mon_free.close();
    mon_df.close();
    mon_redis.close();
    mon_riak.close();
}

#endif
```

Arquivo client.cpp

```
#include <unistd.h>
#include <fstream>
#include <utils.h>

using namespace std;

int main(int argc, char* argv[]) {
```

```
    if (argc < 3) {
        cerr << "Usage: " << argv[0] << " FOLDER FOLDER_NFS [ITERATIONS]"
<< std::endl;
        return 1;
    }

    std::string folder = argv[1];
    std::string folder_nfs = argv[2];
    int iterations = 1;
    if (argc > 3) {
        istringstream ss(argv[3]);
        if (!(ss >> iterations)) {
            std::cout << "Invalid iterations number " << argv[3] << ".
Ignoring, will run 1 iteration." << std::endl;
            iterations = 1; // Better make sure
        } else {
            std::cout << "Will run " << to_string(iterations) << "
iterations" << std::endl;
        }
    }
    //from http://stackoverflow.com/a/18027897/3136474
    if (!folder.empty() && folder.back() != '/')
        folder += '/'; //add trailing slash if not present
    if (!folder_nfs.empty() && folder_nfs.back() != '/')
        folder_nfs += '/'; //add trailing slash if not present
    std::string folder_out = folder + ".out/";
    std::string file_cmd = folder_nfs + "exec";
    std::string file_start = folder_nfs + "start";
    std::string file_stop = folder_nfs + "stop";
    std::ofstream fstream;

    std::cout << "NFS mounted on " << folder_nfs << std::endl;
    std::cout << "Creating folder for output on " << folder_out <<
std::endl;
    mkdir(folder_out.c_str(), 7777);
    std::cout << "Checking files on " << folder << std::endl;

    std::string auxCmd = "ls " + folder;
    std::string ls_output = exec(auxCmd.c_str());
    trim(ls_output);
    if (!ls_output.empty()) {
        std::vector<std::string> files = splitNl(ls_output);
        std::cout << files.size() << " file(s) found" << std::endl;

        for(auto const& file : files) {
            // Read commands from file
            std::string full_filename = folder + file;
            ifstream ifstart(full_filename);
            std::string
command( (std::istreambuf_iterator<char>(ifstart) ),
                                (std::istreambuf_iterator<char>()) );
            trim(command);
            ifstart.close();
            remove(full_filename.c_str()); //Delete file
            std::vector<std::string> commands = splitNl(command);
            std::cout << commands.size() << " commands found in file " <<
file << std::endl;

            // Read pre-benchmark commands from file
            full_filename = folder + ".pre" + file;
            ifstream ifpre(full_filename);
```

```
            std::vector<std::string> precommands;
            if (ifpre.good()) {
                std::string
tmpcommand( (std::istreambuf_iterator<char>(ifpre) ),
                                    (std::istreambuf_iterator<char>()) );
                std::string precommand = tmpcommand;
                trim(precommand);

                if (!precommand.empty()) {
                    precommands = splitNl(precommand);
                    std::cout << precommands.size() << " pre-benchmark
commands found" << std::endl;
                }
            } else {
                std::cout << "No pre command detected in " << full_filename
<< std::endl;
            }
            ifpre.close();
            remove(full_filename.c_str()); //Delete file

            // Read clear commands from file
            full_filename = folder + ".clear" + file;
            ifstream ifclear(full_filename);
            std::vector<std::string> clearcommands;
            if (ifclear.good()) {
                std::string
tmpcommand( (std::istreambuf_iterator<char>(ifclear) ),
                                    (std::istreambuf_iterator<char>()) );
                std::string clearcommand = tmpcommand;
                trim(clearcommand);

                if (!clearcommand.empty()) {
                    clearcommands = splitNl(clearcommand);
                    std::cout << clearcommands.size() << " clear commands
found" << std::endl;
                }
            } else {
                std::cout << "No clear command detected in " <<
full_filename << std::endl;
            }
            ifclear.close();
            remove(full_filename.c_str()); //Delete file

            std::cout << "Starting " << file << " commands:" << std::endl;

            for (int iter = 1; iter <= iterations; iter++) {
                std::string run_id = file; //the name of the file is the
name of the folder created with output
                if (iterations > 1) {
                    run_id += "_" + to_string(iter);
                }

                if (precommands.size() > 0) {
                    std::cout << "Executing pre-benchmark commands" <<
std::endl;
                    for(auto const& precmd : precommands) {
                        std::cout << "Executing '" << precmd << "'" <<
std::endl;

                        // Pre commands are executed client side
                        std::string cmd_output = exec(precmd.c_str());
```

```
                        std::cout << "Output: " << cmd_output << std::endl;

                        // // Tells server to execute this command
                        // fstream.open(file_cmd);
                        // fstream << precmd;
                        // fstream.close(); //flushes

                        // int out_wait = 0;
                        // while (true) {
                        //     out_wait++;

                        //     // Don't notice new files in NFS folder if
we don't touch the folder explicitly
                        //     std::string cmdTmp = "ls " + folder_nfs;
                        //     exec(cmdTmp.c_str());

                        //     std::string file_output = folder_nfs +
"output";
                        //     std::ifstream ifcmdoutput(file_output);
                        //     if (ifcmdoutput.good()) {
                        //         std::string
preoutput( (std::istreambuf_iterator<char>(ifcmdoutput) ),
                        //
(std::istreambuf_iterator<char>()) );
                        //         std::cout << "Got output: " << preoutput
<< std::endl;
                        //         ifcmdoutput.close();
                        //         remove(file_output.c_str());

                        //         break;
                        //     } else {
                        //         ifcmdoutput.close();
                        //     }

                        //     if (out_wait >= 600) {
                        //         std::cout << "10 minutes and no output
so far. Going to next command." << std::endl;
                        //         break;
                        //     }
                        //     sleep(1);
                        // }
                    }
                }

                //Monitor all iterations again
                // if (iter == 1) {
                    // Create start file for server-side script
                    fstream.open(file_start);
                    fstream << run_id;
                    fstream.close(); //flushes
                    std::cout << "Monitoring triggered" << std::endl;
                    // Waits for server to start monitoring
                    sleep(1); //== sleep time on file check on server.cpp
                // }

                // Run commands
                std::string cmds_all_output = "";
                for(auto const& cmdFile : commands) {
                    std::string cmd_output = exec(cmdFile.c_str());
                    cmds_all_output += "\n\n" + cmd_output;
                }
```

```
                // if (iter == 1) {
                    std::cout << "Command executed, stopping monitoring..."
<< std::endl;
                    // Create stop file for server-side script
                    fstream.open(file_stop);
                    fstream << "\n"; //this file is not readed
                    fstream.close(); //flushes
                // }

                sleep(3 * 3); //== 3 x sleep time on monitoring loop on
server.cpp

                std::string cmd = "mv " + folder_nfs + run_id + " " +
folder_out;
                std::cout << "Getting output... " << cmd << std::endl;
                exec(cmd.c_str());

                // Writes command output as well
                std::string filename = folder_out + file +
"_client_output.txt";
                // appends all output on a single file
                fstream.open(filename, std::ofstream::out |
std::ofstream::app);
                fstream << cmds_all_output;
                fstream.close();

                if (iterations > 1) {
                    std::cout << "Iteration " << to_string(iter) << " of ";
                }
                std::cout << file << " done. Output on " << folder_out <<
run_id << std::endl;

                if (clearcommands.size() > 0) {
                    std::cout << "Executing clear commands" << std::endl;
                    for(auto const& clearcmd : clearcommands) {
                        std::cout << "Executing '" << clearcmd << "'" <<
std::endl;

                        // Tells server to execute this command
                        fstream.open(file_cmd);
                        fstream << clearcmd;
                        fstream.close(); //flushes

                        int out_wait = 0;
                        while (true) {
                            out_wait++;

                            // Don't notice new files in NFS folder if we
don't touch the folder explicitly
                            std::string cmdTmp = "ls " + folder_nfs;
                            exec(cmdTmp.c_str());

                            std::string file_output = folder_nfs +
"output";
                            std::ifstream ifcmdoutput(file_output);
                            if (ifcmdoutput.good()) {
                                std::string
clearoutput( (std::istreambuf_iterator<char>(ifcmdoutput) ),

(std::istreambuf_iterator<char>()) );
```

```
                                    std::cout << "Got output: " << clearoutput
<< std::endl;
                                    ifcmdoutput.close();
                                    remove(file_output.c_str());

                                    break;
                                } else {
                                    ifcmdoutput.close();
                                }

                                if (out_wait >= 600) {
                                    std::cout << "10 minutes and no output so
far. Going to next command." << std::endl;
                                    break;
                                }
                                sleep(1);
                            }
                        }
                    }
                } //for iterations loop
            }
        } else {
            std::cout << "No files on " << folder << ". Stopping." <<
std::endl;
        }

}
```

Arquivo server.cpp

```
#include <stdio.h>
#include <iostream>
#include <fstream>
#include <string>
#include <ctime>
#include <unistd.h>
#include <monitoring.h>

using namespace std;

int main(int argc, char* argv[]) {

    if (argc < 2) {
        cerr << "Usage: " << argv[0] << " FOLDER_NFS" << std::endl;
        return 1;
    }

    std::string folder = argv[1];
    //from http://stackoverflow.com/a/18027897/3136474
    if (!folder.empty() && folder.back() != '/')
        folder += '/'; //add trailing slash if not present
    std::string file_cmd = folder + "exec";
    std::string file_start = folder + "start";
    std::string file_stop = folder + "stop";
    std::cout << "Waiting for " << file_start << " to show up..." <<
std::endl;

    int i = 0;
    while (true) {
```

```
        i++;
        std::cout << "." << std::flush;
        if (i % 10 == 0) { //echo each 10s
            std::cout << to_string(i) << " seconds waiting for start file"
<< std::endl;
        }

        std::string cmd = "ls " + folder;
        exec(cmd.c_str());

        std::ifstream ifcmd(file_cmd);
        if (ifcmd.good()) {
            std::cout << "Command file " << file_cmd << " detected, reading
command" << std::endl;

            //from http://stackoverflow.com/a/2912614/3136474
            std::string command( (std::istreambuf_iterator<char>(ifcmd) ),
                                 (std::istreambuf_iterator<char>()) );

            std::cout << "Executing command '" << command << "'" <<
std::endl;

            // Runs command
            std::string cmd_output = exec(command.c_str());

            // Writes command output
            std::string filename = folder + "output";
            std::ofstream fstream;
            fstream.open(filename);
            fstream << cmd_output;
            fstream.close();

            std::cout << "Command executed, removing " << file_cmd << "
file." << std::endl;
            ifcmd.close();
            remove(file_cmd.c_str());
        } else {
            ifcmd.close();
        }

        std::ifstream ifstart(file_start);
        if (ifstart.good()) {
            std::cout << "Start file detected, will monitor until " <<
file_stop << " show up" << std::endl;

            //from http://stackoverflow.com/a/2912614/3136474
            std::string
folder_out( (std::istreambuf_iterator<char>(ifstart) ),
                                 (std::istreambuf_iterator<char>()) );
            //start file must have the folder name for recording output
            trim(folder_out);
            if (!folder_out.empty() && folder_out.back() != '/')
                folder_out += '/'; //add trailing slash if not present
            folder_out = folder + folder_out;
            ifstart.close();
            remove(file_start.c_str());
            mkdir(folder_out.c_str(), 7777);

            Monitoring monitor(folder_out);
```

```
            std::cout << "Redis " << (monitor.redis ? "online" : "offline")
<< std::endl;
            std::cout << "Riak " << (monitor.riak ? "online" : "offline")
<< std::endl;
            std::cout << "Started monitoring, output goes on " <<
folder_out << "..." << std::endl;

            int c = 0;
            while (true) {
                c++;
                std::cout << to_string(c) << " monitoring iteration" <<
std::endl;
                //disk performance (I/O)
                //CPU (general)
                //Memory usage
                //SWAP detection

                //Get CPU and disk usage through iostat
                //avg-cpu:  %user   %nice %system %iowait  %steal   %idle
                //Device:  rrqm/s  wrqm/s  r/s  w/s  rMB/s  wMB/s  avgrq-sz
avgqu-sz  await  r_await  w_await  svctm  %util
                monitor.mon_iostat.read_status();

                //Get memory usage through free
                //  total  used  free  shared  buff/cache  available
                // Mem, Swap, Total
                monitor.mon_free.read_status();

                //Get disk usage through df
                // Filesystem  Type  Size  Used  Avail  Use%  Mounted on
                monitor.mon_df.read_status();

                //Only gets DB info each 5 iterations
                if (c % 5 == 0) {
                    //Get Redis info
                    if (monitor.redis) {
                        //https://redis.io/commands/info
                        monitor.mon_redis.read_status();
                    }

                    //Get Riak info
                    if (monitor.riak) {

//http://docs.basho.com/riak/kv/2.2.3/using/reference/statistics-
monitoring/
                        monitor.mon_riak.read_status();
                    }
                }

                monitor.flush(); //flush all to disk

                sleep(3); //@TODO 3 secs?

                std::string cmd = "ls " + folder;
                exec(cmd.c_str());

                if (file_exists(file_stop)) {
                    remove(file_stop.c_str());
                    std::cout << "Stop file detected, stopping monitoring."
<< std::endl;
                    break;
```

```
                }
            }

            monitor.close();
            std::cout << "Monitoring stopped. Waiting for " << file_start
<< " to show up again..." << std::endl;
            i = 0;
        } else {
            ifstart.close();
        }

        // Wait for 1 sec and check file existence again
        sleep(1);
    }

    return 0;
}
```

APÊNDICE E – AMOSTRAS

Quadro 58 - Amostras bancos chave-valor: Redis (A) e Aerospike (B)

Iteração	*RunTime* A	*RunTime* B	*Throughput* A	*Throughput* B	Latência Leitura A	Latência Leitura B	Latência Grav A	Latência Grav B
1	271410	91392	18422,31	54709,38	844,46	399,74	1708,61	408,33
2	241250	91655	20725,39	54552,40	749,07	401,59	1518,11	408,64
3	263549	95357	18971,80	52434,54	817,46	420,25	1658,13	430,37
4	242292	91150	20636,26	54854,64	752,96	410,28	1527,22	418,39
5	240244	93581	20812,17	53429,65	748,71	407,29	1515,78	415,87
6	250102	90851	19991,84	55035,17	777,33	406,20	1577,86	415,70
7	247033	90039	20240,21	55531,49	767,05	403,31	1559,90	411,68
8	228927	93181	21841,02	53659,01	710,37	418,91	1444,73	427,39
9	240457	91197	20793,74	54826,36	745,10	409,40	1518,71	418,97
10	252791	93287	19779,19	53598,04	788,05	410,30	1595,49	418,36
11	239898	94551	20842,19	52881,51	745,15	423,00	1512,58	432,45
12	233246	96623	21436,59	51747,51	724,48	426,07	1469,60	437,36
13	255902	96851	19538,73	51625,69	794,65	418,81	1607,10	430,44
14	231074	95606	21638,09	52297,97	719,42	424,33	1458,49	433,89
15	240513	92951	20788,90	53791,78	749,00	416,40	1511,22	425,29
16	245818	92614	20340,25	53987,52	763,64	411,75	1549,44	421,61
17	262433	96640	19052,48	51738,41	820,56	429,18	1655,57	438,97
18	257855	91416	19390,74	54695,02	801,31	410,92	1622,36	420,45
19	254813	92668	19622,23	53956,06	791,40	411,07	1608,69	422,04
20	260929	94214	19162,30	53070,67	810,06	410,10	1641,00	419,91
21	238706	94995	20946,27	52634,35	740,18	422,95	1505,07	433,70
22	233709	90498	21394,13	55249,84	724,52	406,99	1467,38	415,92
23	242457	91710	20622,21	54519,68	755,92	410,38	1529,61	420,72
24	262438	91622	19052,12	54572,05	816,38	408,32	1653,26	417,75
25	247849	93925	20173,57	53233,96	771,87	414,69	1561,35	422,28
26	249938	98193	20004,96	50920,13	775,39	439,28	1571,20	450,06
27	249666	94285	20026,76	53030,70	776,94	413,83	1575,00	424,33
28	248665	90323	20107,37	55356,89	771,92	401,99	1565,09	410,60
29	240270	94070	20809,92	53151,91	746,68	410,06	1513,74	421,33
30	243501	89757	20533,80	55705,96	756,33	403,33	1533,80	413,35
31	246220	90729	20307,04	55109,17	769,30	406,95	1551,63	418,09
32	240562	92679	20784,66	53949,65	748,64	408,21	1518,91	418,35

Quadro 59 - Amostra bancos orientados a documentos: MongoDB (A) e Couchbase (B)

Iteração	RunTime A	RunTime B	Throughput A	Throughput B	Latência Leitura A	Latência Leitura B	Latência Grav A	Latência Grav B
1	9536384	4906031	1048,62	2038,31	9959,26	6156,02	1708,01	530,62
2	10260849	4765152	974,58	2098,57	10630,22	5931,50	1849,19	513,99
3	10646318	4964784	939,29	2014,19	11360,24	6217,88	1805,12	506,79
4	10823347	4881866	923,93	2048,40	11658,32	5981,68	1843,25	501,80
5	10335963	4780422	967,50	2091,87	11239,63	6009,83	1743,03	510,64
6	11719797	4775869	853,26	2093,86	13435,31	5971,93	1747,89	510,65
7	12153636	4909470	822,80	2036,88	14040,64	6018,21	1726,62	510,44
8	10824383	4799025	923,84	2083,76	12035,68	6010,40	1738,39	506,60
9	10789291	4796261	926,84	2084,96	11801,19	5937,03	1747,81	510,88
10	11340455	4656654	881,80	2147,46	12855,42	5826,60	1684,55	508,49
11	10779857	4874554	927,66	2051,47	12005,81	6058,96	1656,84	511,20
12	11133239	4840354	898,21	2065,96	11533,68	5975,83	2045,40	513,78
13	10738263	4820625	931,25	2074,42	12203,50	6033,67	1634,32	509,52
14	10774239	4840953	928,14	2065,71	11797,82	6053,05	1786,37	505,87
15	10082425	4911009	991,82	2036,24	10972,80	6157,45	1636,36	510,93
16	11281688	5044060	886,39	1982,53	12443,12	6222,15	1840,83	516,37
17	10742427	4869227	930,89	2053,71	11843,60	6041,26	1758,29	510,22
18	10590568	4832588	944,24	2069,28	11300,59	5971,83	1808,06	511,47
19	10351025	4898892	966,09	2041,28	11684,05	6126,63	1574,35	508,40
20	1062082	4940730	941,55	2023,99	11999,53	6177,37	1644,50	511,09
21	11073808	4697839	903,03	2128,64	12033,48	5804,22	1864,86	513,74
22	1100077	4907422	909,03	2037,73	11963,89	6048,79	1825,76	512,97
23	10281214	4939547	972,65	2024,48	10904,55	6119,51	1749,19	513,41
24	10799738	4819107	925,95	2075,07	11829,23	6084,22	1746,88	520,34
25	10353701	4813207	965,84	2077,62	11248,04	5993,72	1747,08	513,53
26	11283805	4848193	886,23	2062,62	11744,65	6099,43	2014,90	519,23
27	10963601	4801652	912,11	2082,62	11501,63	5874,87	1945,74	509,60
28	11545213	4894068	866,16	2043,29	11932,42	6149,15	2084,31	519,05
29	10870566	4837945	919,92	2066,99	11317,95	6006,89	2004,13	516,16
30	9879044	4911433	1012,24	2036,07	10325,77	6114,69	1783,22	516,77
31	11195872	4781017	893,19	2091,61	11948,33	5904,58	1864,80	518,17
32	10762978	4802662	929,11	2082,18	11778,72	6054,71	1793,54	511,87

APÊNDICE F – PROPOSTA INICIAL

CAPÍTULO 1: CONTEXTUALIZAÇÃO DA PESQUISA

1.1 TEMA

Análise e avaliação comparativa do desempenho de *message brokers* e de bancos de dados em memória.

1.1.1 Delimitação do Tema

Estudo do estado da arte, mapeamento das opções de ferramentas com licença para uso gratuito e triagem das tecnologias de *message brokers* (tais como RabbitMQ, Kafka, ActiveMQ e ZeroMQ) e de bancos de dados em memória (por exemplo, Redis, memcached e Voldemort). Após definir as tecnologias que se destacaram nas propriedades entendidas como mais relevantes pelo meio acadêmico, realizar uma avaliação comparativa de desempenho com a geração de cargas de trabalho (*workload*) através da aplicação de *benchmarks*, os quais serão estudados e escolhidos de acordo com o suporte aos *softwares* selecionados.

Alguns dos indicadores de desempenho que serão avaliados em relação ao *hardware* são: uso de CPU e taxa de fragmentação da memória; em relação aos *message brokers* e aos bancos de dados em memória são: *Throughput* de operações ou mensagens, latência de gravação de mensagens e latência de entrega ou leitura

de mensagens. A aplicação dos *benchmarks* selecionados será executada em um ambiente disponibilizado pelo LARCC[2] (Laboratório de pesquisas avançadas para computação em nuvem), e o mesmo será composto de uma única máquina servidora. Será avaliado o desempenho das diferentes tecnologias sob cargas de trabalho compostas de leituras, escritas e exclusões. Mais cenários de avaliação serão elaborados conforme as possibilidades dos *benchmarks* selecionados para avaliação.

O projeto será realizado durante o período de outubro de 2016 até agosto de 2017 pelos acadêmicos Dinei André Rockenbach e Nadine Anderle como trabalho de conclusão do curso de Sistemas de Informação da Sociedade Educacional Três de Maio – SETREM, na área de Análise de Sistemas, com foco na área de Computação de Alto Desempenho.

1.2 OBJETIVOS

1.2.1 Objetivo Geral

A presente pesquisa busca avaliar e analisar o desempenho de tecnologias de *message broker* e banco de dados em memória.

1.2.2 Objetivos Específicos

- Estudar o estado da arte dos *message brokers* e bancos de dados em memória.

- Analisar funcionalidades dos *message brokers* e bancos de dados em memória.

- Pesquisar *benchmarks* aplicáveis em todos os *message brokers* e bancos de dados a serem avaliados.

- Preparar um ambiente para aplicação dos *benchmarks* nas ferramentas avaliadas.

- Aplicar *benchmarks* nos *message brokers* e nos bancos de dados em memória avaliados.

[2] LARCC – Laboratório de pesquisas avançadas para computação em nuvem. Website: http://www.larcc.com.br/

- Estabelecer a correlação entre as métricas ou indicadores de desempenho.

- Escrever e publicar artigo referente a pesquisa.

1.3 JUSTIFICATIVA

Atualmente existe uma gama expressiva de *softwares*, aplicados em todas as áreas da sociedade, que criam expectativas e necessidades de desempenho excepcional em vários cenários. Dentre os cenários e argumentos para essa necessidade é possível citar impactos em modelos de negócios, custos e gestão empresarial, entre outros.

Para atender tais necessidades, surgiram tecnologias que se propõem a complementar as aplicações desenvolvidas, colaborando com a velocidade e capacidade de processamento de dados. Dentre essas tecnologias, é possível citar os *message brokers* e as bases de dados em memória, que permitem às empresas melhorarem seus aplicativos enquanto mantêm o foco nos mercados em que atuam.

Os *message brokers* são aplicativos de grande impacto para a melhoria na escalabilidade e na separação de responsabilidades de aplicações. De acordo com Videla e Williams (2012), estes podem ser considerados um padrão de projeto (*design pattern*, (GAMMA, HELM, *et al.*, 2000)) que abrange várias funcionalidades tais como distribuição de dados, notificações *push*, "*publish/subscribe*", além de processamento assíncrono de dados, operações não-bloqueantes e filas de trabalhos.

Os bancos de dados em memória ou *in-memory database* (IMDB) são sistemas de bancos de dados que utilizam a memória do computador como dispositivo de armazenamento, ao invés do disco utilizado pelos SGBD (Sistema de Gerenciamento de Banco de Dados) tradicionais. Os dados destes sistemas, portanto, residem inteiramente na memória principal do computador (LAKE e CROWTHER, 2013).

Os *benchmarks*, por sua vez, são testes aplicáveis a várias tecnologias, seja de *hardware* ou *software*, a fim de avaliar seu desempenho e compará-los de modo equivalente, obtendo um ponto de referência a partir do qual pode-se classificar os objetos avaliados (OXFORD, 2016). Eles possuem uma sequência de tarefas que o objeto de análise deve executar, a intenção é induzir cada componente até os seus

limites para, posteriormente, obter uma média de desempenho. Após isso tudo é mensurado e documentado, permitindo checar a classificação obtida.

No entanto, o dia-a-dia conturbado de empresas focadas em desenvolvimento de *software* não oferece boas oportunidades para definir as ferramentas que melhor se adaptam ao seu negócio e auxiliem no atendimento de suas demandas. Isto é agravado pelo fato de que o estudo comparativo de ferramentas requer um investimento considerável de tempo e recursos, além da grande dificuldade em encontrar estudos científicos nesta área no Brasil.

Devido a isso, muitas empresas acabam enfrentando os problemas que estas tecnologias se propõem a resolver, sem saber da existência dessas soluções. Problemas tais como paralelismo no processamento de filas, *message brokers* com bases de dados insustentáveis (VIDELA e WILLIAMS, 2012), gerenciamento de cache inteligente e alto desempenho na busca de pequenos pacotes de dados estruturados (CARLSON, 2013) podem ser citados como alguns exemplos.

É preciso considerar também, que uma parcela considerável das empresas que possuem essa realidade são organizações de pequeno a médio porte, as quais possuem algumas limitações quando a variável se trata de investimentos em infraestrutura. Tendo em vista esse cenário, e com o objetivo de melhor definir o escopo da pesquisa se optou em delimitar o ambiente de aplicação dos testes a uma única máquina servidora.

Levando em consideração os fatos já apresentados, esta pesquisa se propõe a contribuir com a entrega deste comparativo científico, a fim de apresentar as tecnologias de *message brokers* e bancos de dados em memória, e auxiliar essas organizações na escolha das que melhor se adaptam a suas necessidades. Isso colabora para que as pessoas possam usufruir de aplicações cada vez mais performáticas, e que as organizações se utilizem das melhorias alcançadas com a aplicação das tecnologias como diferencial competitivo no mercado.

1.4 PROBLEMA

Dentre os *message brokers* e os bancos de dados em memória estudados, quais oferecem melhor desempenho no quesito velocidade no processamento de dados?

1.5 HIPÓTESES

- O desempenho dos bancos de dados em memória é estatisticamente diferente entre as tecnologias avaliadas.

- O desempenho dos *message brokers* é estatisticamente diferente entre as tecnologias avaliadas.

1.6 VARIÁVEIS

- *Benchmarks*

- Métricas de desempenho

- Ferramentas de *message broker*

- Ferramentas de banco de dados em memória

1.7 METODOLOGIA

1.7.1 Abordagem

Conforme apresentado por Lovato (2013), a metodologia de uma pesquisa possibilita aos pesquisadores duas alternativas a serem empregadas. Sendo que a primeira aponta o estudo para o sentido de reunir resultados através do raciocínio indutivo, dedutivo, ou ainda do conjunto hipotético-dedutivo. Por outro lado, o autor também descreve possibilidades qualitativa, quantitativa ou quali-quantitativa.

O método de pesquisa definido foi o dedutivo, pois parte do conhecimento teórico, ou seja, um ente abstrato para algo concreto, obtido através de observações e experimentos no mundo real. A dedução foi complementada pelo método quantitativo pois os dados obtidos são numéricos e resultados de uma análise

estatística. Os métodos quantitativos utilizados ainda serão definidos após a escolha de quais *benchmarks* serão aplicados e sobre quais ferramentas, uma vez que podem variar de acordo com as saídas que os testes irão retornar.

Deste modo serão aplicados *benchmarks* para avaliar as hipóteses levantadas, em relação ao desempenho da velocidade no processamento de dados das tecnologias de *message brokers* e de banco de dados em memória. Posteriormente à aplicação dos testes com o modelo definido será preciso classificar e analisar os dados coletados.

1.7.2 Procedimentos

Quanto aos métodos de procedimento, o presente trabalho utiliza-se da pesquisa bibliográfica, aplicação de *benchmarks* para coleta de dados e pós coleta compilação e análise dos dados.

A pesquisa bibliográfica, de acordo com a obra de Macedo (1994), pode ser descrita como a busca de informações literárias, utilizando fontes que estão relacionadas ao problema de estudo, podendo estas fontes serem *web sites*, revistas, artigos, livros, enciclopédias, entre outros. Também será realizado um mapeamento sistemático das tecnologias em bases de dados a serem definidas a partir de *strings* de busca (ou *search strings*).

O trabalho de conclusão de curso utilizou o procedimento de pesquisa bibliográfica, com o objetivo de adquirir o embasamento necessário para a busca do preenchimento da lacuna encontrada no problema.

A pesquisa ainda empregou o procedimento experimental pois esse permite operar e modificar as variáveis independentes e ainda medir as variáveis dependentes (LOVATO, 2013). Ainda quando a pesquisa possui abordagem quantitativa a coleta de dados é feita com base em uma trajetória, que pode ser subdividida em: Planejamento, Execução, Coleta de Dados e Análise e Interpretação dos Resultados (STORCK , GARCIA, *et al.*, 2006).

Para validar a primeira hipótese, de que o desempenho dos bancos de dados em memória é estatisticamente diferente entre as tecnologias avaliadas, serão comparadas as métricas obtidas da aplicação dos *benchmarks* nos bancos de dados

em memória e será estabelecido o percentual de diferença entre as distintas tecnologias.

Do mesmo modo, para validar a segunda hipótese, de que o desempenho dos *message brokers* é estatisticamente diferente entre as tecnologias avaliadas, serão aplicados os *benchmarks* nos *message brokers* e serão comparadas as métricas obtidas, para posteriormente estabelecer o percentual de diferença entre as tecnologias.

1.7.2.1 Técnicas

Durante a execução deste trabalho de conclusão de curso o mesmo será amparado por algumas técnicas, podendo destacar a documentação e observação. A documentação se encaixa no momento em que se transcreve os resultados obtidos através da observação, a fim de resguardá-los. A observação será empregada durante a execução dos *benchmarks* para realizar a obtenção dos indicadores.

1.8 ORÇAMENTO

Para a execução deste projeto de foram previstos alguns investimentos financeiros, esta previsão orçamentária está descriminada no Quadro 1. O mesmo está apresentado em 4 colunas, onde a primeira apresenta a descrição do material, a segunda coluna a quantidade prevista, a terceira coluna representa o valor unitário de cada item e a quarta e última coluna o valor total baseado no valor unitário e na quantidade estimada.

Quadro 1 - Previsão orçamentária

Material necessário	Unidade de Medida	Quantidade	Valor(R$) unitário	Total (R$)
Capa e encadernação das entregas parciais	Cópia preta	15	0,12	1,80
Encadernação	Capa	1	120,00	120,00
Impressão do projeto	Cópia preta	60	0,12	7,20
Impressão do relatório	Cópia preta	800	0,12	96,00
Deslocamento	Quilômetro	660	0,50	330,00
Horas de pesquisa e desenvolvimento	Horas	440	25,00	11.000,00
Ambiente estimativa	Meses	3	77,00	231,00
Total em R$				**11.786,00**

Para estimativa do valor relacionado ao ambiente, foram avaliados os orçamentos oferecidos pela Amazon AWS e Google Cloud. O ambiente previsto teria 8 núcleos de processamento, 30 GiB de memória RAM, e disco SSD de 375GB.

1.9 CRONOGRAMA

O cronograma representado no Quadro 2 e Quadro 3 demonstra as etapas que devem ser percorridos durante o desenvolvimento do projeto. Eles estão estruturados em 2 colunas principais, na primeira estão descritas as atividades, e a segunda coluna principal representa o período em que se pretende executar as atividades. A coluna que representa o período ainda é subdividida em colunas secundárias com os meses nos quais o projeto será desenvolvido.

Quadro 2 - Cronograma de Atividades Propostas 2016

	2016					
Atividades	**Out.**		**Nov.**		**Dez.**	
	1	2	1	2	3	4
Elaboração do projeto		X	X			
Desenvolvimento da justificativa		X	X			
Elaboração da metodologia		X	X			
Conclusão do projeto			X			
Entrega do projeto				X		
Referencial teórico			X	X		
Apresentação do projeto do Trabalho de Conclusão de Curso				X		
Alinhamentos para aplicação do TCC				X	X	
Entrega final do projeto, proposta e declaração de autenticidade					X	
Exame						

Quadro 3 - Cronograma de Atividades Propostas 2017

Atividades	2017											
	Jan.		Fev.		Mar.		Abr.		Mai.		Jun.	
	1	2	1	2	1	2	1	2	1	2	1	2
Continuidade Referencial Teórico.	■	■	■	■								
Estudo da arte dos *message brokers* e banco de dados em memória.	■	■										
Analisar funcionalidades dos *message brokers* e bancos de dados em memória.		■	■									
Pesquisar os *benchmarks*, definir quais serão aplicados.				■	■							
Preparar o ambiente para aplicação dos *benchmarks.*						■						
Aplicar os *bechmarks* nos *message brokers* e nos bancos de dados em memória avaliados.						■	■	■	■			
Documentar os resultados obtidos						■	■	■	■	■		
Estabelecer a correlação entre as métricas ou indicadores de desempenho.								■	■	■		
Escrever e publicar artigo referente a pesquisa									■	■		
Entrega final do relatório científico.										■		
Apresentação do trabalho de conclusão de curso												■

Legenda:

Previsto: ■

Realizado: X

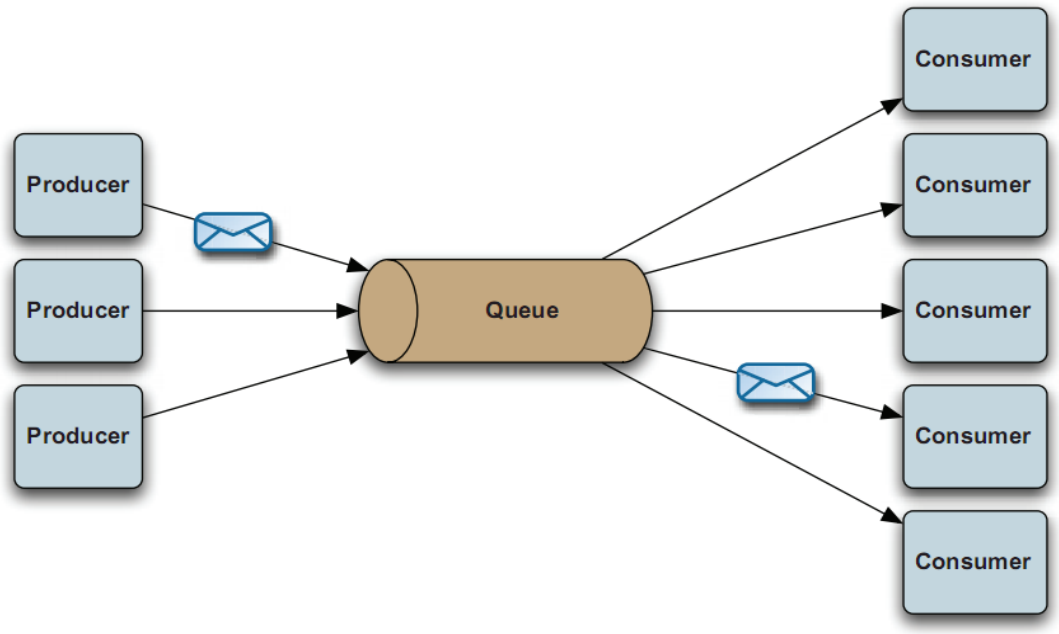
CAPÍTULO 2: REFERENCIAL TEÓRICO

2.1 MESSAGE BROKER

Os *message broker* se encaixam na categoria de *Message Oriented Middleware* (MOM, ou *middlewares* orientados a mensagem, em tradução livre) (IONESCU, 2015).

Um *message broker* é um sistema que encaminha mensagens entre aplicações (VIDELA e WILLIAMS, 2012), facilitando a interoperabilidade, a escalabilidade e a modularização de sistemas complexos. Segundo Ionescu (2015), os *message brokers* são muito utilizados para interconectar softwares independentes, muitas vezes desenvolvidos em linguagens diferentes e rodando em diferentes plataformas.

Figura 1 - Funcionamento dos message brokers



Fonte: (SNYDER, BOSANAC e DAVIES, 2011).

O crescimento da Internet e, consequentemente, da troca de mensagens entre diferentes aplicações, fez da escalabilidade um fator crítico no projeto de aplicações (VIDELA e WILLIAMS, 2012). Os *message brokers* surgem como uma solução para facilitar esta escalabilidade, oferecendo separação de responsabilidades e suporte à altos volumes de dados.

Devido à sua grande adoção, surgiram muitas soluções de *message broker* no mercado. Nas próximas seções estão documentadas as características de algumas destas tecnologias, porém durante o estudo da arte podem ser adicionadas outros *softwares* similares.

2.1.1 RabbitMQ

O RabbitMQ é um *message broker* de código aberto desenvolvido na linguagem Erlang, lançado em 2007 e controlado pela empresa Pivotal desde 2013 (IONESCU, 2015). Este *message broker* permite que aplicações diferentes compartilhem dados através de um protocolo comum, ou simplesmente enfileirem tarefas para processamento por trabalhadores distribuídos (VIDELA e WILLIAMS, 2012).

Algumas das funcionalidades oferecidas pelo RabbitMQ incluem persistência em disco, confirmação de entrega, confirmação de publicação e alta disponibilidade, além de roteamento flexível, clusters lógicos, interface de gerenciamento web e vários plug-ins (RABBITMQ).

A comunicação com o RabbitMQ, por sua vez, pode ocorrer através dos protocolos AMQP (versões 0-8, 0-9, 0-9-1 e 1.0), STOMP e MQTT 3.1, porém apenas o AMQP até a versão 0-9-1 é suportado nativamente, sendo que o suporte ao restante dos protocolos ocorre através de plug-ins (RABBITMQ).

2.1.2 Apache Kafka

O Apache Kafka é um *message broker* desenvolvido em linguagem Scala pela LinkedIn e posteriormente doado para a fundação Apache, momento em que seu código fonte foi aberto (TAVEIRA, 2015). Segundo a própria Apache, o Kafka é uma plataforma de streaming distribuída, definição que deixa claro o seu objetivo de ser utilizado em forma de cluster (APACHE KAFKA).

Kreps, Narkhede e Rao (2011), por sua vez, deixam claro que o objetivo que direcionou os trabalhos de desenvolvimento do Kafka foi o processamento rápido de altos volumes de dados de log. Garg (2015) cita alguns casos de uso do Kafka, como LinkedIn, Twitter e Foursquare.

O *broker* Kafka possui funcionalidades como persistência em disco, alto *Throughput*, clusterização, suporte a clientes multiplataformas e entrega de mensagens em tempo real, sendo uma solução para lidar com altos volumes de informações em tempo real e direcioná-los a múltiplos consumidores rapidamente (GARG, 2015). Taveira (2015), porém, destaca que o Kafka não possui suporte à mecanismos de autenticação, o que limita sua utilização em determinados ambientes.

2.1.3 ActiveMQ

O ActiveMQ é um *message broker* de código aberto sob controle da Apache Software Foundation e distribuído sob a licença Apache (IONESCU, 2015).

Esse *message broker* é muito comum em aplicações Java, principalmente por ele mesmo ser desenvolvido em linguagem Java com JMS (*Java Message Service*) e oferecer suporte aos servidores TomEE, Geronimo, JBoss, GlassFish e WebLogic, além de integração com o framework Spring e persistência em banco de dados utilizando JDBC (APACHE ACTIVEMQ). Apesar disso, Snyder, Bosanac e Davies (2011) destacam que existem clientes do ActiveMQ para todas as principais linguagens de programação do mercado.

Ele suporta os protocolos AMQP 1.0, MQTT 3.1, OpenWrite e STOMP (APACHE ACTIVEMQ), porém cada instância do ActiveMQ pode trabalhar com apenas um protocolo por vez devido à limitações do JMS, não havendo a possibilidade de enviar uma mensagem utilizando um protocolo e receber esta mensagem através de outro protocolo (IONESCU, 2015).

2.1.4 ZeroMQ

Diferentemente de sistemas como RabbitMQ e ActiveMQ, o ZeroMQ (também chamado de 0MQ ou zmq) é uma biblioteca de mensageria, oferecendo as ferramentas necessárias para o desenvolvedor criar seu próprio sistema de *message broker* (AKGUL, 2013).

Esta biblioteca atua como um *framework* para controle de concorrência, oferecendo *sockets* capazes de carregar mensagens atômicas através de variados meios de transportes, tais *como inter-processos, intra-processos*, TCP, entre outros, com padrões como *fan-out*, *pub-sub*, distribuição de tarefas e requisição e resposta (HINTJENS, 2013).